Spring MVC

Domine o principal framework web Java

ALBERTO SOUZA

Agradecimentos

Esse é o primeiro livro que escrevo pela Casa do Código e a satisfação é imensa. Ser um dos autores da editora que desburocratizou a criação de livros no país é uma enorme honra. Por conta disso meu primeiro agradecimento vai para Paulo e Adriano, por me darem espaço para escrever. E como já é de praxe na minha vida, vou deixar aqui o agradecimento para Larissa, minha esposa, por aguentar a ausência necessária para escrever um livro.

Por último quero agradecer a você, que tem sede de conhecimento e quer estar sempre aprendendo. São de pessoas assim que o mundo precisa e eu espero, sinceramente, que cada gota de suor direcionado para seu aprendizado se transforme em um passo a mais dado na sua carreira.

Sumário

Capítulo 1

Introdução

Já existem diversos frameworks MVC no mercado e a primeira pergunta que você deve fazer é: existe a necessidade de aprender mais um? Em geral, existem algumas tecnologias que, quando são escolhidas para serem usadas em uma aplicação, não são facilmente reversíveis. O framework MVC é uma delas.

Ele vai fazer parte de uma boa parte da sua rotina de desenvolvimento. Quase sempre que uma nova funcionalidade for pedida para seu projeto, além de implementar a regra de negócio específica, você terá que criar alguma coisa relativa a essa regra na camada web da sua aplicação. E aí, você mais uma vez estará interagindo com o framework escolhido.

Através desse prisma, você deve ser capaz de analisar os pontos positivos e negativos de cada uma das suas opções e tomar a melhor decisão possível.

1.1 Por que o Spring MVC

O Spring é um projeto de longa data, ele ganhou muita notoriedade nos tempos em que a especificação JavaEE ainda era muito burocrática. Tirou proveito disso e tomou de assalto o mercado desde muito tempo atrás, até os dias de hoje.

A ideia dele sempre foi ser uma alternativa ao JavaEE e, para atingir esse objetivo, eles criaram diversos projetos ao redor do módulo principal, que é o de **injeção de dependências**.
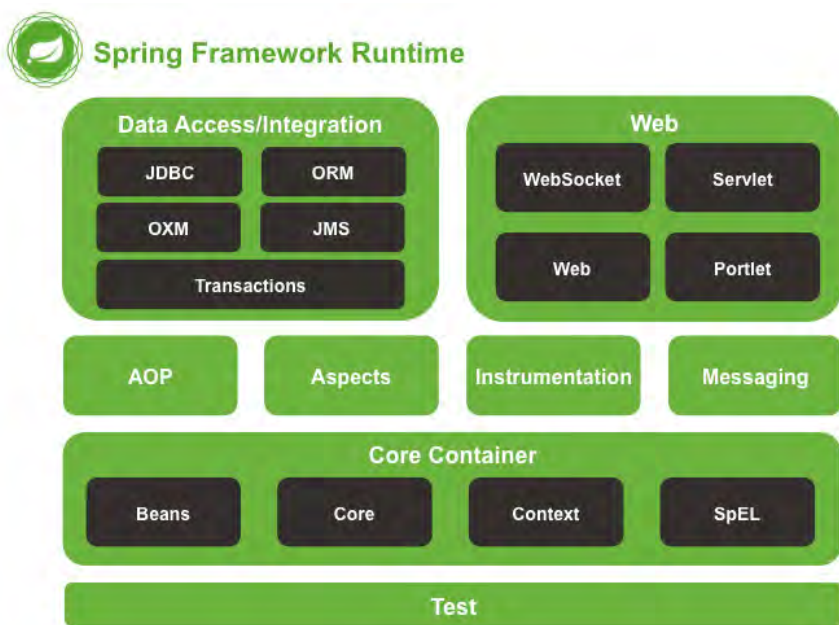
Fig. 1.1: módulos do Spring

Esse é um dos principais motivos para a adoção do Spring MVC, que é justamente o módulo web. Na hora da adoção de um framework, um dos pontos mais relevantes é o que ele lhe entrega de maneira gratuita, e o Spring MVC é completamente integrado com todas as extensões produzidas em cima do próprio Spring. Essa imagem mostra apenas as principais características

do framework. Para conhecer todas as opções, vale a pena acessar o link http://spring.io/projects.

Um outro ponto altamente relevante é o quanto de ajuda você pode obter através da internet. E nesse ponto, os projetos relacionados ao Spring também se beneficiam bastante. Eles têm uma página exclusiva com questões do *StackOverflow*, apenas com as perguntas marcadas com tags relativas ao Spring. Você pode dar um olhada seguindo este: link http://spring.io/questions. Além disso, a ZeroTurnaround, empresa famosa por ferramentas como o JRebel, fez um levantamento dos frameworks MVC mais usados no mercado, o Spring MVC liderou com 40%. Você pode dar uma olhada neste: link http://tinyurl.com/nq8n4zx.

Quase zero de configuração em XML

Atualmente, o Spring suporta que a maioria das suas configurações seja feita de maneira programática, ou seja, você não precisa criar nenhum XML para que seus módulos comecem a funcionar. Esse é mais um ponto que facilitou ainda mais sua adoção. Em algum momento do tempo, pareceu que configurações em XML eram a saída para declarar informações pertinentes ao projeto, mas o ponto é: sempre que você altera uma dessas configurações, é necessário que uma nova instalação do projeto seja gerada. Dada essa condição, por que vamos deixar de escrever código em Java para escrever código em XML? Com uma boa separação de pacotes, conseguimos deixar nossas configurações bem modulares e isso será mostrado no decorrer do livro.

1.2 A margem da especificação

As especificações produzidas pela comunidade Java são de grande valia para as empresas, isso é inegável. Ter uma empresa como a Oracle abalizando as decisões que envolvem a plataforma passa segurança para os tomadores de decisão que querem usar esta linguagem e suas ferramentas no seu projeto.

O ponto negativo, assim como quase todas as decisões que tomamos, é que muitas vezes as especificações demoram para ser criadas e somos obrigados a recorrer a projetos de terceiros, que muitas vezes não estão integrados com as especificações já existentes, para concluirmos alguma tarefa. E é jus-

tamente aí que entra um diferencial dos projetos que usam o Spring como base.

Como eles não precisam seguir as especificações, você já tem diversas opções de integrações que ainda não foram especificadas ou cujas especificações ainda estão imaturas. Alguns exemplos:

- Spring Social para fazer logins sociais;
- Quartz para integração com este agendador de tarefas;
- Spring Data para integração com bancos de dados não relacionais;
- Spring Security para fornecer autenticação e autorização;
- Spring Cache para ter a possibilidade de cachear diversas partes do código.

Essas são extensões que você pode usar nos seus projetos que tenham o Spring como base. Usaremos alguns deles no decorrer do livro. Vale sempre lembrar, nada impede você de tirar proveito do melhor de cada um dos mundos. Caso se identifique mais com um módulo da especificação do que com o do Spring, use-o sem medo algum. Talvez o leitor atento lembre que já existem especificações para agendamento e Cache, e é verdade. Elas só não estão tão maduras quanto as opções Open Source que já existem.

1.3 Comece a aventura

Durante o livro vamos construir uma aplicação muito parecida com a própria Casa do Código. Desenvolveremos funcionalidades que nos levarão desde os detalhes mais básicos do uso do Spring MVC até partes avançadas como:

- customização de componentes de validação e conversão;
- Caches de resultados;
- Processamento assíncrono;
- Testes de integração;

- Detalhes escondidos do processo de internacionalização.

Fique junto comigo e leia cada capítulo com calma; eu sempre lhe direi se é hora de dar uma pausa ou de já pular para o próximo. Espero que você tenha uma grande jornada!

1.4 Público-alvo

Para pessoas que já conheçam um pouco sobre a estrutura de um projeto baseado no Design Pattern MVC. O livro vai ajudar desde pessoas que não conhecem nada do Spring MVC até usuários que já utilizam o framework na prática. Quase todo capítulo vai cobrir detalhes que vão além do uso normal e vai deixá-lo mais crítico em relação ao uso da tecnologia.

1.5 Código-fonte

Todo o código-fonte do livro está disponível no Github, basta acessar o endereço https://github.com/livrospringmvc. Fique à vontade para navegar pelos arquivos do projeto. Os commits foram dividos de acordo com os capítulos do livro, justamente para que você possa acessar o código compatível ao capítulo que esteja sendo lido.

1.6 Pedindo ajuda

Para facilitar a troca de ideias entre os leitores foi criado um fórum de discussão específico para o livro. Basta que você acesse o endereço https://groups.google.com/forum/?hl=pt-BR#!forum/livro-springmvc.

Capítulo 2

Começando o projeto

2.1 Configuração básica e criação do projeto

O objetivo do livro é construir uma aplicação semelhante a da própria Casa do Código. Por ser um site de e-commerce, a aplicação nos fornece a chance de implementar diversas funcionalidades e, além disso, é um domínio que não vai nos causar muitas dúvidas.

A primeira coisa que precisamos fazer é justamente criar e configurar o mínimo necessário para ter nossa aplicação, com o Spring MVC, rodando corretamente. Como vamos usar o Maven para gerenciar as nossas dependências, somos obrigados a criar o projeto seguindo uma estrutura determinada por essa ferramenta. O layout das pastas, como de costume, vai seguir o seguinte formato:
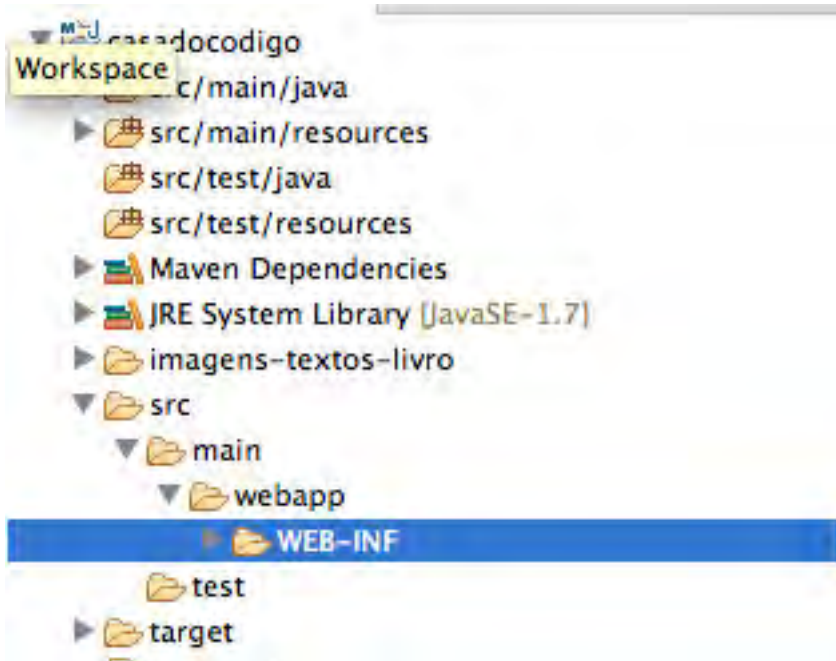
Fig. 2.1: Estrutura de pastas

Provavelmente você já deve ter feito esta atividade algumas vezes, então, para mudar um pouco a sua rotina, vou utilizar um modo um pouco diferente de criar o projeto. Usaremos um projeto da Red Hat chamado **Jboss Forge**, que pode ser baixado neste link http://forge.jboss.org/download. O Jboss Forge fornece vários comandos prontos para a criação de projetos. Após baixado, extraia o arquivo em uma pasta de sua preferência. Acesse o terminal do seu computador e digite o seguinte comando:

```
$caminhoParaSuaInstalacao/bin/forge
```

Isso abrirá uma aplicação no próprio console.

```
JBoss Forge, version [ 2.13.0.Final ] - JBoss, by Red Hat, Inc. [ http://forge.jboss.org ]
[java]$
```
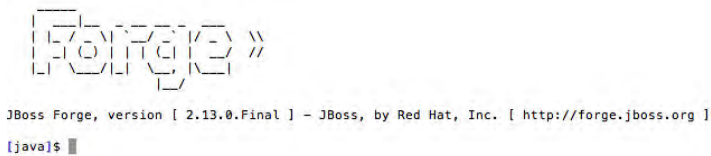
Fig. 2.2: Console do forge

Agora basta que digitemos o comando responsável pela criação do projeto.

```
project-new --named casadocodigo
```

Pronto! Isso é suficiente para você criar o seu projeto baseado no Maven. A estrutura de pastas já está pronta, só precisamos realizar as devidas configurações.

Dependências necessárias

Precisamos adicionar todas as dependências necessárias para que seja possível começarmos a trabalhar no projeto. Para isso, vamos ter o arquivo `pom.xml` e adicionamos a tag `<dependencies>`, dentro da qual colocamos cada uma das dependências necessárias para começar o seu projeto usando o Spring MVC.

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>4.1.0.RELEASE</version>
    </dependency>


    <dependency>
        <groupId>org.apache.tomcat</groupId>
        <artifactId>tomcat-servlet-api</artifactId>
        <version>7.0.30</version>
        <scope>provided</scope>
```

```
    </dependency>
    <dependency>
        <groupId>javax.servlet.jsp</groupId>
        <artifactId>jsp-api</artifactId>
        <version>2.1</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>javax.servlet.jsp.jstl</groupId>
        <artifactId>jstl-api</artifactId>
        <version>1.2</version>
        <exclusions>
            <exclusion>
                <groupId>javax.servlet</groupId>
                <artifactId>servlet-api</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>org.glassfish.web</groupId>
        <artifactId>jstl-impl</artifactId>
        <version>1.2</version>
        <exclusions>
            <exclusion>
                <groupId>javax.servlet</groupId>
                <artifactId>servlet-api</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
    <!-- Logging -->
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId>
        <version>1.6.1</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>jcl-over-slf4j</artifactId>
        <version>1.6.1</version>
```

```
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
            <version>1.6.1</version>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>1.2.16</version>
            <scope>runtime</scope>
        </dependency>
</dependencies>
```

Perceba que foram muitas dependências e nem todas foram necessariamente de projetos ligados diretamente ao Spring MVC. Não perca tempo analisando minuciosamente cada uma delas. Lembre-se sempre: você está em um projeto usando a linguagem Java; se tiver pouca dependência não tem graça.

Agora que o `pom.xml` está configurado corretamente, chegou a hora de importá-lo para a sua *ide* favorita. Durante o livro, usaremos o **Spring Tools Suite**, que é basicamente um Eclipse com alguns plugins específicos para projetos baseados no Spring. Caso prefira usar o seu Eclipse comum, não há nenhum problema.

Vamos importar o projeto como um **Maven Project**; assim, para cada nova dependência necessária, basta alterarmos o `pom.xml` e o próprio Eclipse se encarregará de atualizar o nosso *classpath*.
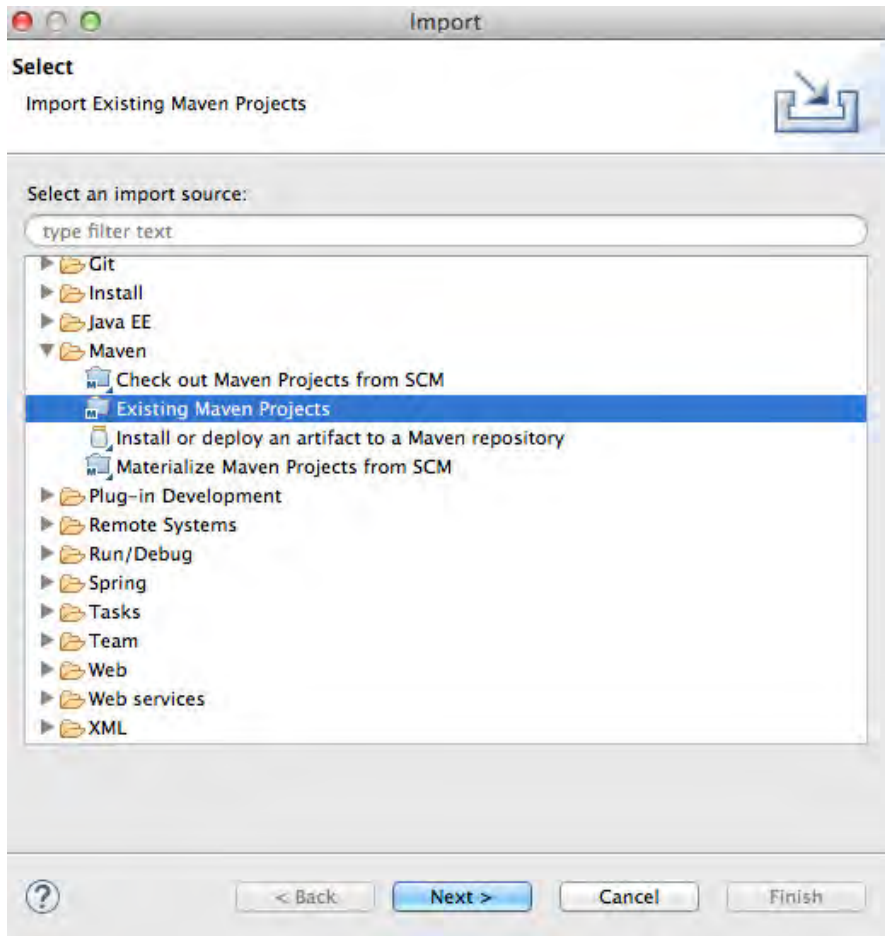
Fig. 2.3: Tipo de importação de projeto

Clicando no `next`, será exibido o *wizard* de importação. Basta escolher o caminho onde seu projeto foi criado.
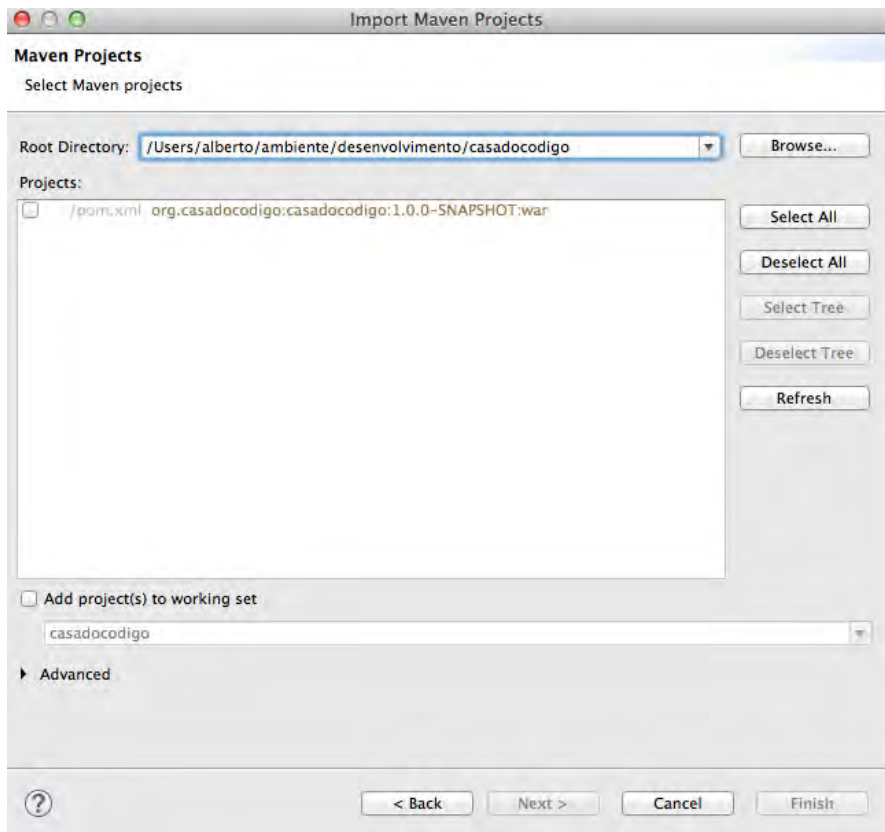
Fig. 2.4: Tela de importação do projeto maven

Pronto, agora temos nosso projeto importado. O último passo é adicioná-lo a um servidor, para que possamos realizar nossos testes e ver como a aplicação vai andando. Para isso, vamos seguir o caminho padrão de criação de servidores dentro do Eclipse.
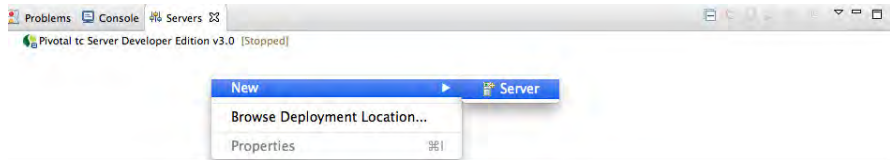
Fig. 2.5: Adicione um novo servidor

Select the server type:
type filter text
Apache
Tomcat v3.2 Server
Tomcat v4.0 Server
Tomcat v4.1 Server
Tomcat v5.0 Server
Tomcat v5.5 Server
Tomcat v6.0 Server
Tomcat v7.0 Server
Tomcat v8.0 Server
Publishes and runs J2EE and Java EE Web projects and server configurations to a local Tomcat server.
Server's host name: localhost
Server name: Tomcat v8.0 Server at localhost
< Back
Next >
Cancel
Finish

Fig. 2.6: Escolha da versão 7 para cima
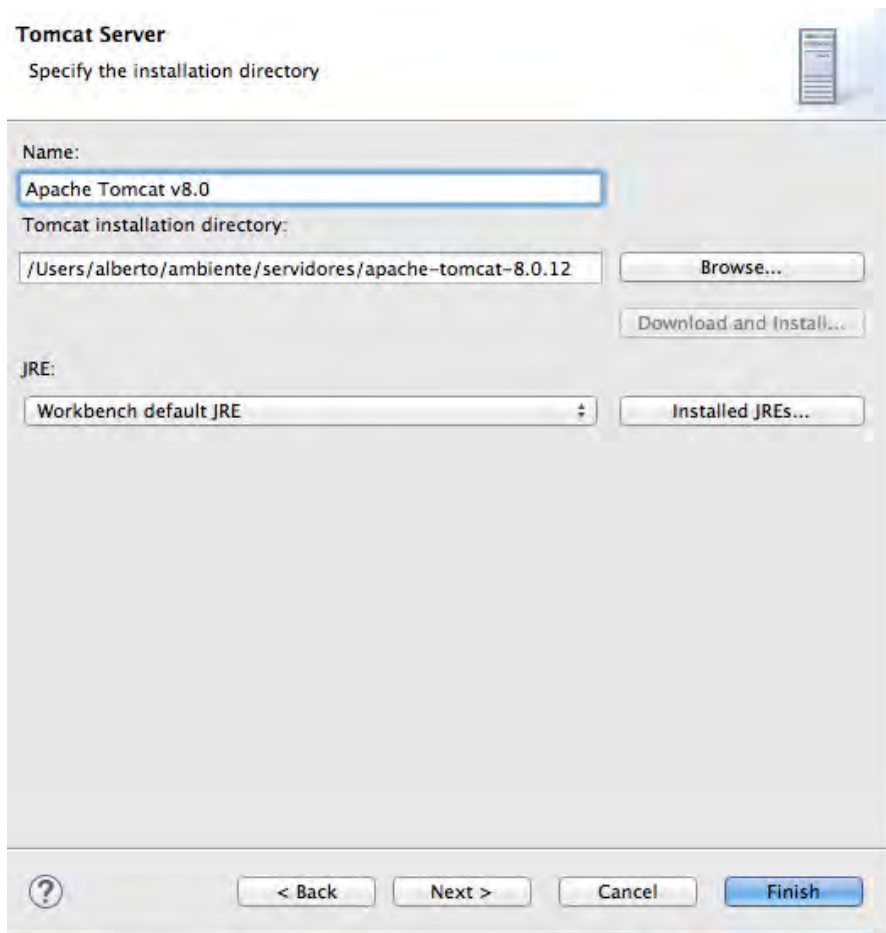
Fig. 2.7: Aponte o caminho de instalação
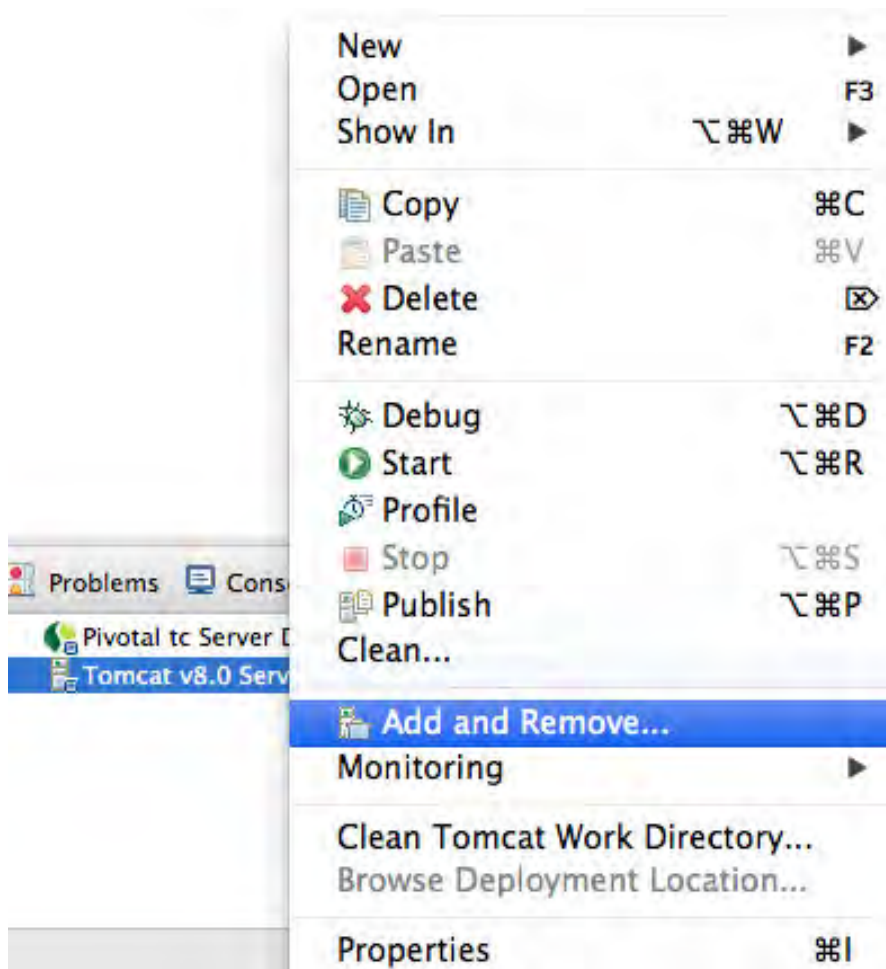
Fig. 2.8: Opção de adicionar projeto
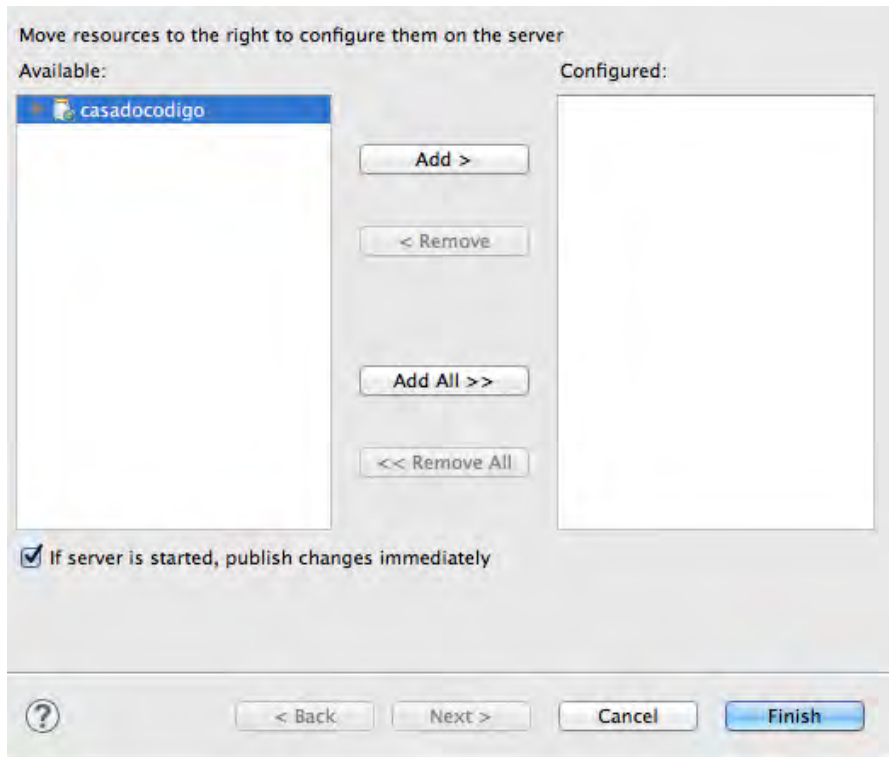
Fig. 2.9: Adicione o projeto ao servidor

Para finalizarmos, vamos subir o servidor e ver se tudo funciona corretamente.

```
Jan 12, 2015 11:36:00 AM org.apache.catalina.core.AprLifecycleListener init
INFO: The APR based Apache Tomcat Native library which allows optimal performance in production environments was not found on the java.library.path: /Users/alberto/Library/
Jan 12, 2015 11:36:00 AM org.apache.tomcat.util.digester.SetPropertiesRule begin
WARNING: [SetPropertiesRule]{Server/Service/Engine/Host/Context} Setting property 'source' to 'org.eclipse.jst.jee.server:casadocodigo' did not find a matching property.
Jan 12, 2015 11:36:00 AM org.apache.coyote.AbstractProtocol init
INFO: Initializing ProtocolHandler ["http-nio-8080"]
Jan 12, 2015 11:36:00 AM org.apache.tomcat.util.net.NioSelectorPool getSharedSelector
INFO: Using a shared selector for servlet write/read
Jan 12, 2015 11:36:00 AM org.apache.coyote.AbstractProtocol init
INFO: Initializing ProtocolHandler ["ajp-nio-8009"]
Jan 12, 2015 11:36:00 AM org.apache.tomcat.util.net.NioSelectorPool getSharedSelector
INFO: Using a shared selector for servlet write/read
Jan 12, 2015 11:36:00 AM org.apache.catalina.startup.Catalina load
INFO: Initialization processed in 1082 ms
Jan 12, 2015 11:36:00 AM org.apache.catalina.core.StandardService startInternal
INFO: Starting service Catalina
Jan 12, 2015 11:36:00 AM org.apache.catalina.core.StandardEngine startInternal
INFO: Starting Servlet Engine: Apache Tomcat/8.0.12
Jan 12, 2015 11:36:02 AM org.apache.catalina.core.ApplicationContext log
INFO: No Spring WebApplicationInitializer types detected on classpath
Jan 12, 2015 11:36:02 AM org.apache.coyote.AbstractProtocol start
INFO: Starting ProtocolHandler ["http-nio-8080"]
Jan 12, 2015 11:36:02 AM org.apache.coyote.AbstractProtocol start
INFO: Starting ProtocolHandler ["ajp-nio-8009"]
Jan 12, 2015 11:36:02 AM org.apache.catalina.startup.Catalina start
INFO: Server startup in 1790 ms
```

Fig. 2.10: Console indicando que o servidor subiu

2.2 Acessando o primeiro endereço

Com tudo configurado, chegou a hora de realmente começarmos a produzir alguma coisa! O nosso primeiro desafio é levar o nosso usuário para a home da Casa do Código. O endereço que queremos que ele digite é o `/home`. Em um cenário mais comum, a URL que leva o usuário para a home é o próprio `/`. Faremos essa alteração no final do capítulo. Neste momento, caso o usuário digite http://localhost:8080/casadocodigo/home, ele receberá um erro do servidor indicando que esse endereço não pode ser atendido por ninguém.

Respondendo a URLs

Em geral, com o Spring MVC, assim como com quase qualquer outro framework MVC, sempre que quisermos responder a uma URL, teremos que criar uma classe responsável por isso.

```
package br.com.casadocodigo.loja.controllers;

public class HomeController {

}
```

Apenas por convenção, é sugerido que o nome da classe termine com o sufixo **Controller**. Agora precisamos ensinar ao Spring MVC que essa classe, efetivamente, é responsável por atender requisições vindas de um

cliente, nesse caso o navegador. Para isso, vamos adicionar a annotation `@Controller` nela.

```
package br.com.casadocodigo.loja.controllers;

@Controller
public class HomeController {

}
```

E como vamos fazer a lógica de carregamentos de produtos para serem exibidos na `home`? Nada mais comum do que criarmos um método.

```
@Controller
public class HomeController {

    public void index(){
        //aqui ainda vamos carregar os produtos.
    }
}
```

Agora imagine o seguinte cenário. Você provavelmente terá diversos controllers, com as mais variadas responsabilidades e métodos. Como o Spring MVC vai saber qual método deve ser chamado para cada uma das URLs? É para isso que serve a annotation `@RequestMapping`:

```
@Controller
public class HomeController {

    @RequestMapping("/home")
    public void index(){
        //aqui ainda vamos carregar os produtos.
        System.out.println("Carregando os produtos");
    }
}
```

Essa última configuração que acabamos de fazer é também conhecida como **binding**. Fizemos a ligação entre uma rota e o método responsável por tratá-la. O problema é que mesmo que você tente acessar o endereço agora,

ainda continuará recebendo o código de erro 404, indicando que o servidor não consegue lidar com essa URL.

2.3 Habilitando o Spring MVC

Por mais que já tenhamos criado nosso primeiro controller, ainda não fizemos nenhuma configuração para habilitar que o Spring MVC reconheça essas classes. Caso você puxe pela sua memória, vai lembrar que, para conseguir responder a URLs, em projetos web seguindo a especificação de Servlets, é necessário ter no mínimo um **Filter** ou uma **Servlet** configurada. E é justamente esse passo que vamos realizar a partir de agora.

Configuração programática

A Servlet responsável por tratar **todas** as requisições que chegam para o Spring MVC é a **DispatcherServlet**. Precisamos justamente dela para que tudo comece a funcionar. A maneira tradicional seria fazer essa configuração através do arquivo `web.xml` mas, como vocês podem ver, esse arquivo nem foi criado. Tudo será feito através de classes Java. A primeira tarefa é criar uma classe que seja filha de `AbstractAnnotationConfigDispatcherServletInitializer`.

```
package br.com.casadocodigo.loja.conf;

//imports omitidos

public class ServletSpringMVC extends
 AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return null;
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return null;
```

```
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] {"/"};
    }

}
```

O objetivo de herdar dessa classe é justamente não ter que ficar fazendo o registro do `DispatcherServlet` na mão. Temos alguns métodos interessantes, vamos começar pelo `getServletMappings`. Aqui é onde você diz qual é o padrão de endereço que vai ser delegado para o Servlet do Spring MVC. Caso fôssemos usar o `web.xml`, essa configuração seria equivalente ao `<url-mapping>`.

E como o Spring MVC vai saber quais controllers devem ser mapeados e quais outras classes devem ser carregadas pelo container do próprio Spring? Para essa questão, usamos o método `getServletConfigClasses`. Nele, será retornada uma ou mais classes responsáveis por indicar quais outras classes devem ser lidas durante o carregamento da aplicação web.

```
@Override
protected Class<?>[] getServletConfigClasses() {
    return new Class[]{AppWebConfiguration.class};
}
```

Agora vamos criar a classe `AppWebConfiguration`.

```
package br.com.casadocodigo.loja.conf;

@EnableWebMvc
@ComponentScan(basePackageClasses={HomeController.class})
public class AppWebConfiguration {
}
```

Perceba que é uma classe um tanto quanto estranha. Tem um monte de annotation, mas nenhum código. O objetivo principal dela é expor para a Servlet do Spring MVC quais são as classes que devem ser lidas e carregadas.

Neste exato momento, a principal annotation aqui é a `@ComponentScan`. Através dela indicamos quais pacotes devem ser lidos. Perceba que passamos como parâmetro a classe `HomeController` para que o Spring leia o pacote dela. Dessa forma, não passamos uma String e evitamos um possível erro de digitação.

A annotation `EnableWebMvc` não é necessária agora, mas como ela já habilita várias funcionalidades que serão usadas na nossa aplicação web, já vamos deixá-la aí. Para você não ficar se perguntando quais seriam essas funcionalidades, segue uma lista com algumas delas:

- Conversão de objetos para XML
- Conversão de objetos para JSON
- Validação usando a especificação
- Suporte a geração de RSS

Uma última observação importante é que devemos manter essas classes em um pacote separado, focado justamente em classes que servem apenas de configuração. Por isso estamos usando o pacote `br.com.casadocodigo.loja.conf`.

Agora que temos tudo configurado, caso você tente acessar o endereço `/home`, infelizmente deverá receber a seguinte tela de erro.

Fig. 2.11: View não foi encontrada

2.4 Configurando a pasta com as páginas

O erro que vimos na seção anterior indica que o Spring MVC não achou uma página responsável por montar o HTML relativo ao request. Puxando pela memória, o fluxo padrão de uma aplicação MVC é: Cliente faz request, Servlet principal atende e delega para um controller da aplicação e, finalmente, uma página é montada para responder a requisição. Por default, o Spring MVC segue justamente essa sequência.

Como o retorno do nosso método é **void**, ele procurou por uma página com nome em branco e ela não foi encontrada. O primeiro passo para tentarmos contornar o erro é retornar uma String, indicando o caminho da página.

```
@RequestMapping("/")
public String index(){
    System.out.println("Carregando os produtos");
    return "hello-world.jsp";
}
```

Caso você tente acessar agora, vai receber outro erro.

Fig. 2.12: Exception gerada pela falta de um resolvedor de View

O Spring MVC não sabe onde procurar por essa página, precisamos ensiná-lo! Para fazer isso, precisamos configurar um objeto cuja responsabilidade é informar onde as páginas devem ser encontradas. E sempre que falamos em configuração, pelo menos por enquanto, estamos nos referindo à classe `AppWebConfiguration`.

```
@Bean
public InternalResourceViewResolver
 internalResourceViewResolver() {
  InternalResourceViewResolver resolver =
              new InternalResourceViewResolver();
  resolver.setPrefix("/WEB-INF/views/");
  resolver.setSuffix(".jsp");
  return resolver;
}
```

A classe `InternalResourceViewResolver`, como pode ser lido no próprio código, guarda as configurações da pasta base e do sufixo que devem ser adicionados para qualquer caminho que seja retornado por métodos dos controllers. A annotation `@Bean` indica para o Spring que o retorno desse método deve ser registrado como um objeto gerenciado pelo *container*. Em geral, no Spring, esses objetos são chamados de Beans. Para deixar tudo correto, vamos apenas alterar o retorno do método `index`, do nosso primeiro controller.

```
@Controller
public class HomeController {

    @RequestMapping("/")
    public String index(){
        System.out.println("Carregando os produtos");
        return "hello-world";
    }
}
```

Agora basta que você crie uma página chamada `hello-world` dentro da pasta `WEB-INF/views` e tudo funcionará corretamente.

Talvez você esteja se perguntando o motivo de deixar a página dentro da pasta WEB-INF. A grande sacada é você quase que obrigar os desenvolvedores a escreverem métodos no controller para cada funcionalidade. Desse jeito, o seu projeto está se protegendo de ter lógicas jogadas diretamente dentro das páginas.

2.5 Um pouco por dentro do framework

Um ponto que muitas vezes diferencia "usuários" do framework de "entendedores" é justamente ter o conhecimento do que acontece por dentro da ferramenta. Por exemplo, como o Spring MVC faz com que a nossa classe `ServletSpringMVC` seja carregada? Onde está o código de carregamento da `DispatcherServlet`? Essas são perguntas que, por mais que não tenham a ver com o seu código de negócio em si, podem ser importantes para entender algum fluxo do framework.

O web.xml é desnecessário

Desde a versão 3 da especificação de Servlets, justamente para facilitar a configuração dos frameworks, foi criado um outro tipo de arquivo chamado de `web-fragment.xml`. A ideia é que esse arquivo possa ser deixado dentro da pasta `META-INF`, no `jar` da biblioteca. Dessa forma, quando o servidor web for iniciado e a aplicação for carregada, esse arquivo vai ser lido e os servlets ou filtros que lá estiverem serão criados e registrados dentro do servidor. Não por acaso, o `jar` do Spring MVC possui este arquivo:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-fragment ....>

    <name>spring_web</name>
    <distributable/>

</web-fragment>
```

Analisando, ele é bem simples. Tem um nome e usa a tag `distributable`, apenas para dizer que a aplicação usando esse arquivo pode ser distribuída entre vários servidores. Só que, um ponto interessante é: onde está a configuração do servlet do Spring MVC?

Um pouco mais a fundo na especificação

O web-fragment é apenas para dizer que o Spring MVC pode ser notificado sobre o início da aplicação web. Contudo, desta maneira ele ainda não está fazendo nada. Um outro arquivo interessante, dentro do `jar`, é

o `javax.servlet.ServletContainerInitializer`. Ele também fica dentro da pasta `META-INF` e o conteúdo é o que segue:

```
org.springframework.web.SpringServletContainerInitializer
```

Esse arquivo, especificamente com este nome, é lido pelo servidor web em questão e a classe configurada é carregada.

```
@HandlesTypes(WebApplicationInitializer.class)
public class SpringServletContainerInitializer implements
ServletContainerInitializer {


    @Override
    public void onStartup(Set<Class<?>> webAppInitializerClasses,
    ServletContext servletContext)
            throws ServletException {

        List<WebApplicationInitializer> initializers = new
        LinkedList<WebApplicationInitializer>();

        if (webAppInitializerClasses != null) {
            for (Class<?> waiClass : webAppInitializerClasses) {
                if (!waiClass.isInterface() &&
                !Modifier.isAbstract(waiClass.getModifiers()) &&
                        WebApplicationInitializer.class.
                        isAssignableFrom(waiClass)) {
                    try {
                        initializers.add(
                        (WebApplicationInitializer)
                        waiClass.newInstance());
                    }
                    catch (Throwable ex) {
                        throw new ServletException("Failed to
                        instantiate WebApplicationInitializer
                        class", ex);
                    }
                }
            }
```

```
        }

        if (initializers.isEmpty()) {
            servletContext.log("No Spring
            WebApplicationInitializer types detected on
            classpath");
            return;
        }

        AnnotationAwareOrderComparator.sort(initializers);
        servletContext.log("Spring WebApplicationInitializers
        detected on classpath: " + initializers);

        for (WebApplicationInitializer initializer :
        initializers) {
            initializer.onStartup(servletContext);
        }
    }

}
```

A annotation `@HandlesTypes` recebe como argumento um conjunto de interfaces, cujas classes filhas são passadas como parâmetro para o método `onStartup`. Caso você esteja curioso, verá que a nossa classe `ServletSpringMVC` é filha da interface `WebApplicationInitializer`, justamente a que foi passada para o `@HandleTypes`!

2.6 Conclusão

Este capitulo foi um pouco longo e, para ser bem sincero, até simples. O problema é que foi necessária muita configuração para um simples *hello world*. A parte boa dessa história é que você passou por uma das etapas mais difíceis no aprendizado de qualquer framework, o setup inicial. Agora tudo tende a ser mais fluido! Por isso, eu indico que você continue a leitura. No próximo capítulo já começaremos a acessar o banco de dados para cadastrar e listar os produtos da loja.

Capítulo 3

Cadastro de produtos

Passada a fase de configuração, chegou a hora de começarmos a implementar algumas das funcionalidades da loja. Uma das features mais básicas, porém importante, é o cadastro de livros, já que sem livros não existem compras.

3.1 Formulário de cadastro

Vamos começar a nossa implementação justamente pela parte mais próxima do nosso usuário final, que é o formulário de novo produto.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Cadastro de produtos</title>
</head>
```

```
<body>

    <form method="post">
        <div>
            <label for="title">Titulo</label>
            <input type="text" name="title" id="title"/>
        </div>
        <div>
            <label for="description">Descrição</label>
            <textarea rows="10" cols="20" name="description"
            id="description">
            </textarea>
        </div>
        <div>
            <label for="pages">Número de paginas</label>
            <input type="text" name="pages" id="pages"/>
        </div>
        <div>
            <input type="submit" value="Enviar">
        </div>
    </form>

</body>
</html>
```

Não tem nada de mais, apenas um HTML normal. Só para constar, nosso modelo de livro não precisa ser muito complexo, tem apenas um título, descrição e o número total de páginas.

Um detalhe importante que ficou faltando: para onde vamos enviar as informações cadastradas? Ainda não criamos nada que trate essa lógica, mas já podemos imaginar para qual URL queremos enviar. Para isso, vamos alterar a declaração da tag `<form>` e vamos adicionar o atributo `action`.

```
<form method="post" action="/casadocodigo/produtos">
```

Aqui vale uma observação. Foi usado o nome do contexto de maneira *hard coded*, enquanto a estratégia mais usada no mercado é o uso da tag `<c:url>`, que vem no pacote de tags padrão JSTL. O motivo dessa escolha é

simplesmente o da facilidade do entendimento do código. A tag, misturada com as aspas dos atributos das tags `html`, muitas vezes confundem mais que ajudam, pelo menos durante a leitura do livro.

E onde vamos criar esse arquivo? A nossa configuração atual diz que devemos criar nossas views dentro da pasta `WEB-INF/views`. Como estamos lidando com o cadastro de produtos, vamos manter uma organização e criar uma outra subpasta, chamada `produtos`. O caminho final do arquivo deve ser `WEB-INF/views/products/form.jsp`. Usaremos essa tática para criação das próximas páginas, durante o decorrer do livro.

3.2 Lógica de cadastro

Neste exato momento, se o nosso usuário tentar cadastrar um produto, ele vai receber o famoso erro de página não encontrada. Afinal de contas, ainda não criamos o controller responsável por essa funcionalidade. Vamos começar a resolver este problema agora.

```
br.com.casadocodigo.loja.controllers;

@Controller
public class ProductsController {

    @RequestMapping("/produtos")
    public String save(){
        System.out.println("Cadastrando o produto");
        return "products/ok";
    }
}
```

Até aqui, nada que já não tenhamos visto. Criamos um controller e um método, ambos com as anotações necessárias para o Spring MVC reconhecer o controller e também fazer o *binding* entre a rota e o respectivo método. E onde está o produto que estamos tentando cadastrar?

Recebendo objetos como parâmetros

Aqui não vamos ter nenhuma novidade. Todo framework que se preze deve permitir que o objeto seja montado e passado como parâmetro no mé-

todo.

```
br.com.casadocodigo.loja.controllers;

@Controller
public class ProductsController {

    @RequestMapping("/produtos")
    public String save(Product product){
        System.out.println("Cadastrando o produto "+product);
        return "products/ok";
    }
}
```

Não temos a classe `Product`, chegou a hora de criá-la:

```
br.com.casadocodigo.loja.models;
public class Product {
    private String title;
    private String description;
    private int pages;

    //getters e setters
}
```

Perceba que os nomes dos campos do nosso formulário batem exatamente com os nomes dos *setters* que serão gerados para nós.

- title => setTitle
- description => setDescription
- pages => setPages

É dessa forma que o Spring MVC consegue popular o objeto em questão. Caso você queira, adicione um `toString` na classe `Product`. Dessa forma, a impressão vai mostrar exatamente os valores que foram preenchidos no formulário.

URL para chegar ao formulário

Um ponto que o leitor mais atento deve estar se perguntando é: qual é o endereço que o usuário vai digitar para chegar no formulário? Como a JSP está dentro da pasta `WEB-INF`, ela não está visível para ser acessada pelo lado de fora do servidor.

A solução é justamente criar mais um método no controller, para levar o usuário para a página em questão.

```
package br.com.casadocodigo.loja.controllers;

@Controller
public class ProductsController {
    //resto do código

    @RequestMapping("/produtos/form")
    public String form(){
        return "products/form";
    }
}
```

O método vazio, muitas vezes, é algo que incomoda o programador. Podemos até nos perguntar: por que não deixamos a JSP fora da pasta `WEB-INF` e fazemos o acesso direto? O problema dessa abordagem é que está sendo dada uma brecha para que alguma lógica, além da necessária para a visualização, seja colocada dentro da página. Um pouco mais para a frente, ainda neste capítulo, vamos precisar carregar os tipos de livros que podem ser vendidos. Caso não tivesse sido criado esse método, onde esse código seria colocado? Podemos dizer que esse é um mal que veio para nosso bem.

3.3 Gravando os dados no banco de dados

Por mais que nosso formulário já esteja um tanto quanto funcional, ainda não estamos efetivamente gravando as informações passadas. Para melhorar essa parte, vamos começar a efetuar as alterações necessárias a partir do nosso controller.

A primeira coisa é começar a usar um *DAO* responsável pelo acesso aos dados referente à classe `Product`.

Lembrete sobre o DAO

O Data Access Object é apenas uma classe cujo objetivo é isolar o acesso aos dados de um determinada parte do sistema. No nosso exemplo, vamos precisar gravar, listar e carregar um produto. Não queremos misturar essa parte de acesso à infraestrutura com nossas lógicas.

O código do método `save` deve ficar parecido com o que segue:

```
public String save(Product product){
    productDAO.save(product);
    return "products/ok";
}
```

E de onde vem essa instância da classe `ProductDAO`? Uma das opções é que a gente mesmo tenha criado esse objeto, na mão.

```
private ProductDAO productDAO = new ProductDAO();

public String save(Product product){
    productDAO.save(product);
    return "products/ok";
}
```

O problema é que, provavelmente, o construtor do `ProductDAO` vai precisar receber algum objeto que represente a conexão com o banco de dados e, nesse caso, vamos ter que começar a controlar essas dependências na mão. Essa é uma ótima parte para usarmos o Spring MVC, pois como ele é completamente integrado com o Spring, podemos pedir que o Spring crie esse objeto para gente. O nosso único trabalho é indicar que precisamos receber **injetado** uma instância do `ProductDAO`.

```
public class ProductsController {

    ...

    @Autowired
```

```
    private ProductDAO productDAO;

    public String save(Product product){
        productDAO.save(product);
        return "products/ok";
    }
}
```

A annotation `@Autowired` é justamente a responsável por indicar os pontos de injeção dentro da sua classe.

Nesse exato momento, nosso código não compila, já que não criamos ainda a classe `ProductDAO`, assim como não existe o método `save`. Chegou a hora de resolvermos essa parte do nosso código.

```
package br.com.casadocodigo.loja.daos;

@Repository
public class ProductDAO {

    @PersistenceContext
    private EntityManager manager;

    public void save(Product product){
        manager.persist(product);
    }

}
```

Aqui estamos usando a JPA para realizar a parte de acesso efetivo ao nosso banco de dados. Na próxima seção já vamos tratar de como configurá-la. Voltando ao nosso `ProductDAO`, ela é uma classe bem normal e o único detalhe ao qual ficar atento é em relação ao uso da annotation `@Repository`. Para que o Spring carregue essa classe e gerencie todo seu ciclo de vida, é necessário que seja usada uma anotação. No nosso exemplo, estamos usando a `@Repository`, apenas para indicar que essa classe, além de ser gerenciada pelo Spring, é também responsável pelo acesso a dados. Existem alguns tipos possíveis de annotations que podemos usar quando queremos indicar que uma classe seja gerenciada pelo container.

- `@Component`: a semântica envolvida é que essa classe é um bean do Spring.
- `@Respository`: a classe é responsável pelo acesso a dados.
- `@Controller`: para indicar que essa classe interage com os requests vindos da web.
- `@Service`: para indicar que a classe representa um componente intimamente ligado a alguma regra de negócio do sistema.

O ideal é sempre usar a annotation que carrega a semântica mais próxima ao uso da classe. Além disso, o seu uso de forma correta ainda facilita a vida do Spring. Por exemplo, quando você utiliza a `@Controller` está indicando que os métodos da classe em questão são responsáveis por tratar requisições e isso faz com que eles sejam escaneados pelo Spring.

3.4 Configurando a JPA com o Hibernate

Estamos quase conseguindo gravar o produto, só falta agora configurarmos a JPA para que nosso acesso ao banco de dados fique completo. Para fazer isso, precisamos seguir alguns passos.

Dependências para a JPA, Hibernate e o Mysql

É necessário que sejam adicionadas as dependências ao arquivo `pom.xml`, dessa forma o Maven pode baixar os `jars` necessários.

```
<project ....
    <dependencies>
        ...
        <!-- configuracao jpa e driver -->
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-entitymanager</artifactId>
            <version>4.3.0.Final</version>
        </dependency>
```

```
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-core</artifactId>
            <version>4.3.0.Final</version>
        </dependency>

        <dependency>
            <groupId>org.hibernate.javax.persistence</groupId>
            <artifactId>hibernate-jpa-2.1-api</artifactId>
            <version>1.0.0.Final</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-orm</artifactId>
            <version>4.1.0.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.15</version>
        </dependency>
    </dependencies>
</project>
```

Como importamos nosso projeto como um *Maven Project*, o próprio eclipse já vai baixar e atualizar o nosso classpath com as novas dependências!

Mapeando a entidade

Com as dependências devidamente configuradas, chegou a hora de fazermos as configurações necessárias para que os objetos da classe `Product` possam ser salvos. Um passo importante é ensinar a implementação da JPA, nosso caso o Hibernate, que a classe `Product` vai representar uma tabela no banco de dados.

```
@Entity
public class Product {
```

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Integer id;
private String title;
@Lob
private String description;
private int pages;
```

Aqui usamos algumas annotations específicas da JPA para realizar esses ensinamentos.

- `@Entity`: indica que a classe vai virar uma tabela.
- `@Id`: indica que o atributo em questão é a chave primária.
- `@GeneratedValue`: indica a maneira como vai ser gerada a chave primária.
- `@Lob`: indica que o atributo em questão vai ser salvo como Clob ou Blob no banco de dados.

Caso você não seja familiarizado com a JPA, fique tranquilo, vamos ficar apenas no básico dela, nosso foco aqui é o Spring MVC. De todo jeito, a documentação do Hibernate é uma fonte de estudos.

Isolando a configuração da JPA

Agora, como para qualquer acesso ao banco de dados, é necessário que informemos algumas detalhes importantes como:

- Driver que deve ser usado para o acesso;
- Login e Senha do banco de dados instalado;
- URL de acesso ao banco;
- Configurações da implementação da JPA.

Para realizar essas configurações, vamos criar uma nova na classe, no pacote `br.com.casadocodigo.loja.conf`.

```
public class JPAConfiguration {

    @Bean
      public LocalContainerEntityManagerFactoryBean
          entityManagerFactory(){
         LocalContainerEntityManagerFactoryBean em = new
             LocalContainerEntityManagerFactoryBean();
         em.setDataSource(dataSource());
         em.setPackagesToScan(new String[]
             { "br.com.casadocodigo.loja.models" });

         JpaVendorAdapter vendorAdapter =
             new HibernateJpaVendorAdapter();
         em.setJpaVendorAdapter(vendorAdapter);
         em.setJpaProperties(additionalProperties());

         return em;
      }

      @Bean
      public DataSource dataSource(){
         DriverManagerDataSource dataSource =
             new DriverManagerDataSource();
         dataSource.setDriverClassName("com.mysql.jdbc.Driver");
         dataSource.setUrl(
             "jdbc:mysql://localhost:3306/casadocodigo");
         dataSource.setUsername( "root" );
         dataSource.setPassword( "" );
         return dataSource;
      }


      private Properties additionalProperties() {
         Properties properties = new Properties();
         properties.setProperty("hibernate.hbm2ddl.auto",
                                 "update");
```

```
        properties.setProperty("hibernate.dialect",
            "org.hibernate.dialect.MySQL5Dialect");
        properties.setProperty("hibernate.show_sql", "true");
        return properties;
    }
}
```

Aconteceu bastante coisa nesse trecho de código, mas nada muito diferente do que você já conhece. O método `dataSource()` serve para configurarmos os parâmetros de conexão com o banco de dados. Você vai usar esse mesmo estilo de configuração mesmo que esteja usando JDBC puro, ou qualquer outro framework de acesso a dados.

Já o método `entityManagerFactory` precisa de um pouco mais de carinho. Ele cria alguns objetos que são importantes para o nosso entendimento do que realmente está acontecendo. A classe `LocalContainerEntityManagerFactoryBean` é a abstração do arquivo `persistence.xml`, que geralmente é necessário para termos a JPA funcionando no nosso projeto. A classe `HibernateJpaVendorAdapter` representa a nossa escolha de implementação da JPA que, no nosso projeto, será o Hibernate. O módulo de *ORM* do Spring suporta ainda, nativamente, o EclipseLink e a OpenJPA.

A annotation `@Bean`, usada em cima dos métodos, é para indicar que os objetos criados por eles vão ser gerenciados pelo Spring e podem ser injetados em qualquer ponto do código.

Agora os últimos detalhes de configuração. O primeiro é que precisamos informar ao servlet do Spring MVC que ele também deve carregar essa classe, para que todos esses métodos possam ser lidos e os objetos disponibilizados.

```
public class ServletSpringMVC extends
    AbstractAnnotationConfigDispatcherServletInitializer {


    ...

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class[]{AppWebConfiguration.class,
```

```
        JPAConfiguration.class};
    }
}
```

O segundo é que na **AppWebConfiguration**, é necessário informar que o pacote da classe **ProductDAO** deve ser lido, para que todas as classes que possuam alguma das annotations de componentes do Spring, possam ser carregadas.

```
@EnableWebMvc
@ComponentScan(basePackageClasses = { HomeController.class,
    ProductDAO.class })
public class AppWebConfiguration extends WebMvcConfigurerAdapter{
    ...
}
```

@PersistenceContext

Na classe `ProductDAO`, para pedirmos a injeção do *EntityManager*, foi usada a annotation `@PersistenceContext`. Essa annotation vem da própria especificação JPA. Geralmente, ela é usada dentro dos servidores JavaEE, mas nada impede de outro framework qualquer fazer uso da mesma. Perceba que o nome `PersistenceContext` tem todo um valor semântico: ele realmente indica que a classe quer receber ali um *EntityManager*. Como isso vai ser implementado, isso é problema de cada ferramenta.

3.5 Habilitando o controle transacional

Caso você seja um leitor um pouco mais ansioso, já deve ter tentado gravar um novo produto no banco de dados e, provavelmente, não está muito contente com o resultado que recebeu.
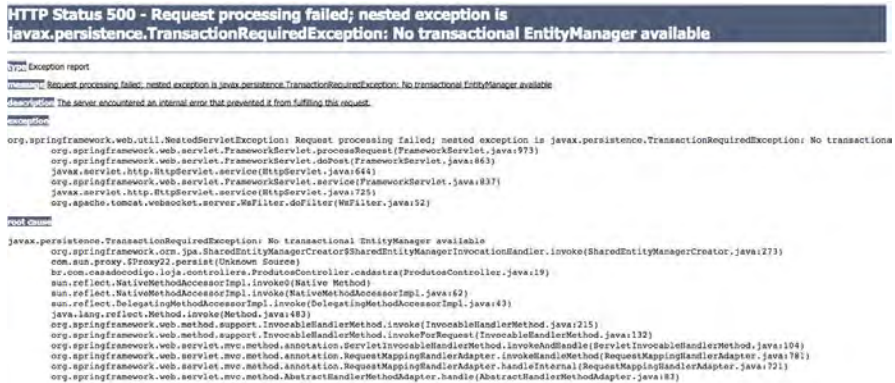
Fig. 3.1: Exception por falta de um gerenciador de transações

Como a própria exception informa, não temos ainda um *EntityManager* capaz de realizar operações transacionais. Para a nossa sorte, isso é apenas mais um passo que devemos configurar. Vamos adicionar mais um método e uma annotation na classe `JPAConfiguration`.

```
@EnableTransactionManagement
public class JPAConfiguration {
    ...

    @Bean
    public PlatformTransactionManager transactionManager
        (EntityManagerFactory emf){
      JpaTransactionManager transactionManager =
          new JpaTransactionManager();
      transactionManager.setEntityManagerFactory(emf);
      return transactionManager;
    }
}
```

`@EnableTransactionManagement` é annotation que indica que agora vamos usar o controle transacional do Spring.

Além de indicar que queremos o controle transacional, precisamos informar por qual implementação vamos optar. A classe `PlatformTransactionManager` é justamente o ponto central do

Spring para esse controle. No nosso caso está sendo usada a implementação para a JPA. Ainda poderíamos, por exemplo, usar a implementação direta para o Hibernate ou até para o JDBC.

Além desse passo, precisamos informar que os métodos da `ProductsController` precisam de transação. Para isso, basta que seja adicionada a annotation `@javax.transaction.Transactional` em cima da classe.

```
@Controller
@Transactional
public class ProductsController {
 ...
}
```

Assim como a `@PersistenceContext`, a `@Transactional` pertence a uma especificação do JavaEE, só que nesse caso, a JTA.

Pronto, agora o cadastro de produtos deve estar funcionando corretamente. Salve alguns produtos e depois consulte no banco de dados, para ver se está tudo lá!

3.6 Conclusão

Neste capítulo, navegamos por alguns pontos importantes do projeto. Agora já sabemos como receber objetos montados a partir de parâmetros do formulário, assim como vimos como realizar toda configuração necessária para usar a JPA dentro do nosso projeto. Caso não esteja ainda cansado, já leia o próximo capítulo, pois vamos incrementar o cadastro, fazer a listagem e mais algumas coisas.

Capítulo 4

Melhorando o cadastro e a listagem

4.1 Recebendo uma lista de valores no formulário

O nosso formulário já faz o cadastro básico de um produto, só que na Casa do Código o mesmo livro pode ser vendido em formatos diferentes e a nossa implementação atual não suporta isso. O primeiro ponto a ser alterado é que nossa classe `Product` deve suportar uma lista de preços e seus respectivos formatos.

```
...

@Entity
```

```
public class Product {
    ...
    @ElementCollection
    private List<Price> prices = new ArrayList<Price>();

    //getters e setters
}
```

Agora precisamos criar a classe `Price`.

```
package br.com.casadocodigo.loja.models;

@Embeddable
public class Price {

    @Column(scale = 2)
    private BigDecimal value;
    private BookType bookType;

    //getters e setters

}
```

Para termos os tipos de livros bem definidos, vamos usar uma **enum**, no caso a `Booktype`.

```
public enum BookType {
    EBOOK,PRINTED,COMBO
}
```

Agora, se rodarmos a aplicação, por conta da criação do `EntityManagerFactory`, o nosso banco de dados já vai ser alterado e a nova tabela já vai ser criada.

Modificando nosso formulário

Com a classe já alterada, precisamos fazer as devidas modificações na página que contém o formulário de cadastro do produto.

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
...

<c:forEach items="${types}" var="bookType" varStatus="status">
    <div>
        <label for="price_${bookType}">${bookType}</label>
        <input type="text" name="prices[${status.index}].value"
            id="price_${bookType}"/>
        <input type="hidden"
            name="prices[${status.index}].bookType"
            value="${bookType}"/>
    </div>
</c:forEach>
```

Dê uma respirada, já que esse pequeno trecho a mais no formulário já contém muita coisa. O ponto mais importante aqui é que, para passar uma lista de valores do formulário para o controller do Spring MVC, é necessário o uso do `[]`. Usamos o atributo `varStatus` para conseguir ter acesso ao índice atual do loop e, com isso, ir construindo os inputs. Na situação atual, o `html` gerado seria como o que segue:

```
<div>


        <div>
            <label for="price_EBOOK">EBOOK</label>
            <input type="text" name="prices[0].value"
                id="price_EBOOK"/>
            <input type="hidden" name="prices[0].bookType"
                value="EBOOK"/>
        </div>

        <div>
            <label for="price_PRINTED">IMPRESSO</label>
            <input type="text" name="prices[1].value"
                id="price_PRINTED"/>
            <input type="hidden" name="prices[1].bookType"
                value="IMPRESSO"/>
        </div>
```

```
        <div>
            <label for="price_COMBO">COMBO</label>
            <input type="text" name="prices[2].value"
                id="price_COMBO"/>
            <input type="hidden" name="prices[2].bookType"
                value="COMBO"/>
        </div>

</div>
```

Como a nossa classe `Product` possui agora os métodos de acesso ao atributo `prices`, o próprio Spring MVC vai popular o nosso objeto. Essa regra serve para qualquer situação que você queira passar uma lista de informações do formulário para o controller. O uso do [**índice**] será sempre necessário. Por exemplo, como desafio, o leitor pode fazer um método na classe `ProductsController` que receba uma lista de livros a serem cadastrados.

4.2 Disponibilizando objetos na view

Um ponto que talvez tenha chamado a atenção foi a realização do `forEach` sobre a variável `types`. Afinal de contas, onde ela foi criada e como ela foi disponibilizada? Essa vai ser justamente uma das finalidades do método `form`, na classe `ProductsController`. Chegamos até a comentar, no capítulo anterior, que usaríamos este método para não dar a chance de nenhum usuário chegar à página do formulário diretamente pela JSP. O argumento foi justamente que, caso fosse necessário disponibilizar objetos para a view, seria melhor que esse código ficasse dentro de uma classe e não de uma página. Para esse tipo de tarefa, o Spring MVC oferece a classe `ModelAndView`.

```
@RequestMapping("/form")
public ModelAndView form(){
    ModelAndView modelAndView =
        new ModelAndView("products/form");
    modelAndView.addObject("types", BookType.values());
    return modelAndView;
}
```

A classe `ModelAndView` possui métodos que nos permitem ir adicionando objetos que serão disponibilizados na view. Para quem quiser fazer o paralelo, por trás dos panos o Spring MVC vai pegar as chaves e valores passados para esse objeto e repassar para o request atual, algo parecido com o que segue:

```
Map<String,Object> modelMap = modelAndView.getModelMap();
Set<String> keys = modelMap.keySet();
for(String key : keys){
    Object value = modelMap.get(key);
    request.setAttribute(key,value);
}
```

No fim das contas, é o método `setAttribute` que garante que os objetos estarão acessíveis na view.

E para qual view o Spring MVC deve ir, já que agora está sendo devolvido um objeto do tipo `ModelAndView`? É justamente para isso que serve o construtor recebendo a String.

```
new ModelAndView("products/form");
```

Vale lembrar que, por conta da configuração do `InternalResourceViewResolver` estamos apenas passando a pasta e o nome da página, que estão dentro de `WEB-INF/views`.

Outra maneira de passar objetos para a view

Em vez de retornar um objeto do tipo `ModelAndView`, podemos receber como parâmetro do método, um objeto do tipo `Model`.

```
public String form(Model model){
    model.addAttribute("types", BookType.values());
    return "products/form";
}
```

Aqui é mais uma questão de gosto. Na minha opinião, retornar um objeto do tipo `ModelAndView` deixa mais claro que o método está disponibilizando objetos para a página.

Um outro ponto a ser observado é o recebimento desse parâmetro. Essa é outra possibilidade de o Spring injetar objetos nas classes.

4.3 Listando os produtos

Outra funcionalidade importante na Casa do Código é a de listar os livros. O interessante dessa implementação é que não precisaremos usar nada de novo, e servirá como revisão de alguns dos conceitos que foram discutidos até aqui.

Mais uma vez, vamos começar pensando na página, que terá o nome de `list.jsp` e ficará, como já era esperado, na pasta `WEB-INF/views/product`:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core"  prefix="c"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
    charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
    <table>
        <tr>
```

```
            <td>Titulo</td>
            <td>Valores</td>
        </tr>
        <c:forEach items="${products}" var="product">
            <tr>
                <td>${product.title}</td>
                <td>
                    <c:forEach items="${product.prices}"
                      var="price">
                        [${price.value} - ${price.bookType}]
                    </c:forEach>
                </td>
            </tr>
        </c:forEach>
    </table>
</body>
</html>
```

Para o usuário chegar a essa página, terá que passar pelo nosso controller. Logo, já vamos criar o método responsável por isso.

```
@Controller
@Transactional
public class ProductsController {
    //resto do código

    @Autowired
    private ProductDAO productDAO;


    @RequestMapping("/produtos")
    public ModelAndView list(){
        ModelAndView modelAndView =
            new ModelAndView("products/list");
        modelAndView.addObject("products", productDAO.list());
        return modelAndView;
    }
}
```

Agora, para esse código compilar, precisamos criar o método `list` na

classe `ProductDAO`.

```
@Repository
public class ProductDAO {

    @PersistenceContext
    private EntityManager manager;

    ...

    public List<Product> list() {
        return manager.createQuery("select distinct(p) from
            Product p join fetch p.prices",Product.class)
            .getResultList();
    }
}
```

Um detalhe que o leitor mais atento pode ter notado é que os métodos `save` e `list`, na classe `ProductsController`, estão com a mesma URL. E agora?

4.4 Melhor uso dos verbos HTTP

Neste momento, caso a URL `/produtos` seja acessada, vamos receber a seguinte exception:

```
java.lang.IllegalStateException: Ambiguous mapping found....
```

O resto da exception foi ocultado, já que o começo dela diz tudo. Temos dois métodos tentando usar o mesmo mapeamento de URLs. Pensando bem nas URLs, elas até fazem sentido:

- `/produtos`, para adicionar um novo produto em uma lista que já existe;
- `/produtos`, para listar os produtos cadastrados.

O problema aqui é que estamos deixando de usar um recurso importante do protocolo HTTP, os verbos.

- `POST`: usado quando existe a necessidade de criação de algum recurso;
- `GET`: usado quando o interesse é o de recuperar alguma informação;
- `DELETE`: como o nome diz, deve ser usado para excluir algum recurso;
- `PUT`: associado com alguma operação de atualização de recursos no servidor.

Estes talvez sejam os verbos mais conhecidos, com destaque para o `post` e o `get`. Por sinal, quando criamos uma classe que herda de `HttpServlet`, temos a opção de implementar cada um deles, separadamente.

O verbo, junto com a URL, serve justamente para indicar o tipo de operação que deve ser realizada no servidor. Por exemplo, se o endereço `/produtos` for acessado através de um `get`, quer dizer que uma listagem está sendo requisitada. Caso seja acessada através de um `post`, está sendo pedido para um novo produto ser criado. Vamos ver como fica a representação dessa teoria em nosso código:

```
...

@RequestMapping(value="/produtos",method=RequestMethod.POST)
public String save(Product product){
    ...
}

@RequestMapping(value="/produtos",method=RequestMethod.GET)
public ModelAndView list(){
    ...
}
```

Pronto! Agora o Spring MVC vai escolher qual método deve ser usado em função do verbo utilizado no acesso ao endereço `/produtos`. Essa é uma ótima prática, tente sempre definir qual verbo deve ser utilizado para acessar cada URL do seu sistema.

4.5 Modularização das URLs de acesso

Ainda sobre os mapeamentos das nossas URLs, perceba que todos os endereços relacionados com o cadastro de produtos começam com `/produtos`.

```
@RequestMapping("/produtos/form")
public String form(){
    return "products/form";
}

@RequestMapping(value="/produtos",method=RequestMethod.POST)
public String save(Product product){
    ...
}

@RequestMapping(value="/produtos",method=RequestMethod.GET)
public ModelAndView list(){
    ...
}
```

Provavelmente, esse padrão vai aparecer em diversos módulos da sua aplicação e, para diminuir essa repetição, o Spring MVC permite que você anote um endereço base diretamente sobre a classe.

```
@Controller
@Transactional
@RequestMapping("/produtos")
public class ProductsController {

    @RequestMapping("/form")
    public String form(){
        return "products/form";
    }

    @RequestMapping(method=RequestMethod.POST)
    public String save(Product product){
        ...
    }

    @RequestMapping(method=RequestMethod.GET)
```

```
    public ModelAndView list(){
        ...
    }
}
```

Perceba que agora, quando necessário, você pode colocar apenas o complemento do endereço.

4.6 Forward x Redirect

Agora que já inserimos e listamos, chegou a hora de melhorar um pouco fluxo entre essas operações. Neste momento, quando um produto é inserido, o usuário é levado para uma página chamada `ok.jsp`, que apenas indica que um novo livro foi cadastrado com sucesso. Nesse tipo de cenário, o fluxo mais indicado é voltar com o usuário para a listagem, talvez mostrando uma mensagem de sucesso.

```
@RequestMapping(method=RequestMethod.POST)
public ModelAndView save(Product product){
    productDAO.save(product);
    return list();
}
```

Esse código implementa justamente o fluxo sugerido. Quando acabamos de salvar um novo produto, invocamos o método `lista`, que é o responsável por listar os produtos e jogá-los para a página `products/list.jsp`. O ponto negativo dessa solução é que o endereço que fica na barra do navegador ainda é o último acessado pelo usuário, que nesse caso foi um `post` para `/produtos`. Caso o nosso cliente aperte um `F5`, o navegador vai tentar refazer a última operação, causando uma nova inserção de produto no sistema.

Fig. 4.1: Problema do forward depois de um post

Observe que o próprio navegador percebe que tem algo de estranho e pergunta se você tem certeza de que quer reenviar os dados. Essa técnica de redirecionamento que acontece apenas do lado do servidor é o que chamamos, no mundo Java, de **forward**. O browser nem sabe o que aconteceu, tanto que, se olharmos no console do Chrome, ele só vai identificar um request.

Fig. 4.2: Chrome tools indica apenas um request

É considerada uma má prática realizar um forward após o usuário ter feito um `post`, justamente por conta do problema da atualização. Para esse cenário, a melhor solução é forçar o usuário a fazer uma nova requisição para a nossa listagem e, dessa forma, permitir que ele atualize a página sem que um novo `post` seja realizado.

```
@RequestMapping(method=RequestMethod.POST)
public String save(Product product){
    productDAO.save(product);
    return "redirect:produtos";
}
```

O prefixo `redirect:` indica para o Spring MVC que, em vez de simplesmente fazer um forward, é necessário que ele retorne o status 302 para o

navegador, solicitando que o mesmo faça um novo request para o novo endereço.
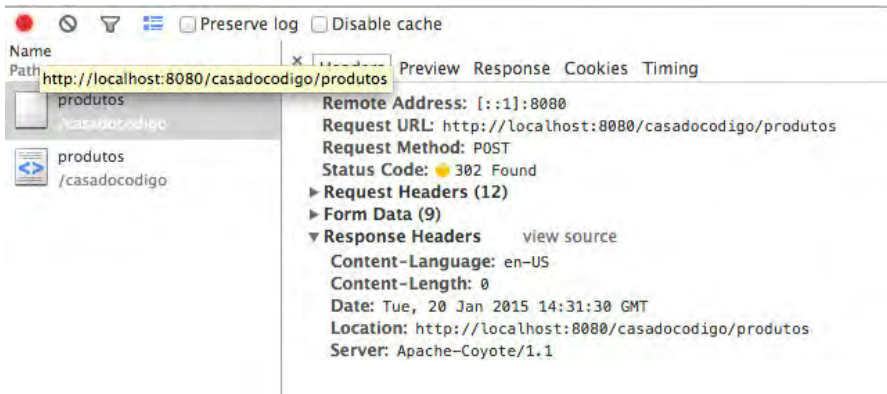
Fig. 4.3: Chrome tools indica dois requests

Perceba que entre os cabeçalhos contidos na resposta existe um chamado **Location**, que informa justamente qual é o endereço ao qual o navegador deve fazer a próxima requisição. Essa técnica, onde fazemos um redirect do lado do cliente logo após um `post`, é um padrão conhecido da web chamado de **Always Redirect After Post** e deve ser sempre utilizado.

4.7 Parâmetros extras nos redirects

Ainda sobre o nosso redirect, geralmente é necessário que indiquemos para o usuário que realmente tudo ocorreu bem. Isso é feito exibindo uma mensagem de sucesso para o cliente em questão.

Uma das maneiras de realizar essa tarefa é utilizando algumas classes que já estudamos.

```
@RequestMapping(method=RequestMethod.POST)
public ModelAndView save(Product product){
    productDAO.save(product);
    ModelAndView modelAndView =
        new ModelAndView("redirect:produtos");
```

```
    modelAndView.addObject("sucesso",
                           "Produto cadastrado com sucesso");
    return modelAndView;
}
```

Como o endereço passado para o `ModelAndView` contém um redirect, o próprio Spring MVC vai pegar os objetos adicionados e passá-los como argumentos da URL `/produtos`.

```
http://localhost:8080/casadocodigo/produtos?
    sucesso=Produto+cadastrado+com+sucesso
```

Essa técnica funciona tranquilamente, mas um dos medos dos desenvolvedores é que algum cliente possa alterar a URL e causar algum dano para o sistema. Outra questão é que os objetos adicionados no `ModelAndView` para um redirect, só podem ser do tipo String. O motivo é que, como já mostramos, eles serão usados como parâmetros da URL.

Outra maneira de passarmos esses objetos é guardando-os na sessão do usuário. O problema é que esses objetos serão mantidos enquanto nosso usuário estiver ativo na aplicação e, na nossa situação, queremos apenas que eles sejam mantidos até o próximo request. Foi justamente para este tipo de cenário que criaram um novo escopo dentro dos frameworks, chamado de **flash**.

```
@RequestMapping(method=RequestMethod.POST)
public String save(Product product,
  RedirectAttributes redirectAttributes){
    productDAO.save(product);
    redirectAttributes.addFlashAttribute("sucesso",
        "Produto cadastrado com sucesso");
    return "redirect:produtos";
}
```

A classe `RedirectAttributes` é justamente a responsável por isso. Todo objeto adicionado nela, através do método `addFlashAttribute`, ficará disponível até o próximo request, sendo acessível de maneira simples através da Expression Language.

```
${sucesso}
```

4.8 Conclusão

Neste capítulo, estudamos mais detalhes sobre como o Spring MVC lida com os formulários. Além disso, vimos como podemos ser mais específicos em relação às nossas URLs, usando os verbos. Para fechar, deixamos nossos endereços mais modulares e ainda discutimos *forwards* e *redirects*, inclusive vendo como funciona a passagem de objetos para os dois modelos de redirecionamentos.

Capítulo 5

Validação e conversão de dados

O nosso cadastro já está funcional, mas estamos deixando de fazer uma coisa básica: validar os dados de entrada. Neste momento, caso um usuário queira cadastrar um novo livro, ele tem a opção de deixar todos os campos em branco.

5.1 Validação básica

A primeira opção de solução para essa situação é escrever o código de validação dentro do próprio método `save`, da classe `ProductsController`.

```
@RequestMapping(method=RequestMethod.POST)
public String save(Product product,
  RedirectAttributes redirectAttributes){

    if(StringUtils.isEmpty(product.getTitle())){
```

```
        //adiciona erro de validação
    }
    if(StringUtils.isEmpty(product.getDescription())){
        //adiciona erro de validação
    }

    productDAO.save(product);
    redirectAttributes.addFlashAttribute("sucesso",
        "Produto cadastrado com sucesso");
    return "redirect:produtos";
}
```

A lógica não tem nada de nada de complicado, na verdade deve ser bem parecida com várias que o leitor já presenciou. Um ponto negativo dessa solução é que o controller, que idealmente deveria apenas ficar chamando lógicas do sistema e controlando qual é o próximo de navegação, agora também está responsável pelo código de validação. Uma outra questão é que o objeto do tipo `Product` pode ser validado em outros lugares do sistema, por exemplo na funcionalidade de salvar vários livros de uma vez. Para contornarmos essas situações, podemos deixar a lógica de validação em uma outra classe, especializada apenas para isso. E como essa é uma funcionalidade requerida em várias aplicações, o Spring já possui um módulo focado apenas em validação, vamos começar a usá-lo.

```
package br.com.casadocodigo.loja.validation;

import org.springframework.validation.Validator;

public class ProductValidator implements Validator{

    @Override
    public void validate(Object target, Errors errors) {
        ValidationUtils.rejectIfEmptyOrWhitespace(errors,
            "title", "field.required");
        ValidationUtils.rejectIfEmptyOrWhitespace(errors,
            "description", "field.required");

        Product product = (Product) target;
```

```
        if(product.getPages() == 0){
            errors.rejectValue("pages", "field.required");
        }
    }

}
```

A regra de validação permanece basicamente a mesma, o mais interessante é a parte que o Spring nos fornece. Primeiramente implementamos a interface `Validator`, que nos obriga a escrever dois métodos:

- `supports`: analisaremos esse método em alguns minutos.
- `validate`: responsável pela validação em si.

O método `validate` recebe como argumento o objeto a ser validado, no caso especificado pelo parâmetro `target` e um outro objeto do tipo `Errors`, onde vamos guardando cada uma das falhas de validação. A classe `ValidationUtils` é um *helper* do Spring Validation para realizar algumas validações básicas. Aqui é interessante analisar a assinatura dos métodos que começam com `reject`.

```
public static void rejectIfEmpty(Errors errors, String field,
    String errorCode) {....}
```

- `Errors errors`: objeto onde será guardado cada um dos erros de validação.
- `String field`: representa o atributo do modelo que deve ser validado.
- `String errorCode`: chave que será usada para buscarmos a mensagem relativa a esse erro.

Um detalhe importante em relação ao parâmetro `field` é que, por exemplo, se quisermos validar o estado dos objetos do tipo `Price`, da nossa classe `Product`, teríamos que usar a seguinte sintaxe: `price.value`. Isso

vale para outros casos também. Se for necessário validar as informações de endereço de uma classe `Usuario`, provavelmente seria passada a String `endereco.rua` e assim por diante.

Para fechar, caso seja preciso fazer uma lógica de validação que não exista na classe utilitária, pode ser usado o método `rejectValue` para adicionar a mensagem na lista de erros. Esse método é da própria classe `Errors`.

```
public class ProductValidator implements Validator{

    @Override
    public void validate(Object target, Errors errors) {
    ....

    Product product = (Product)target;
    if(product.getPages() == 0){
        errors.rejectValue("pages", "field.required");
    }
}
```

Na verdade, isso é o que é feito dentro dos métodos da classe `ValidationUtils`.

Configurando a validação no Controller

Agora que temos nossa classe responsável pela validação de objetos do tipo `Product`, precisamos pedir para o Spring MVC fazer uso dela, no momento da construção do objeto com os dados vindos do formulário.

```
@RequestMapping(method=RequestMethod.POST)
public String save(@Valid Product product,
    RedirectAttributes redirectAttributes){
    ...
}
```

A annotation `@Valid` vem da especificação `Bean Validation`, e é utilizada por diversos frameworks para indicar o disparo do processo de validação. Agora, um pensamento importante: como o Spring MVC vai saber que é para usar a classe `ProductValidator` para validar o objeto do tipo

`Product`? Para resolver isso vamos falar para o Spring MVC qual validator ele deve usar.

```
@Controller
@Transactional
@RequestMapping("/produtos")
public class ProductsController {

    @InitBinder
    protected void initBinder(WebDataBinder binder) {
        binder.setValidator(new ProductValidator());
    }

    ...
}
```

A annotation `@InitBinder` indica que esse método deve ser chamado sempre que um request cair no controller em questão. O nome do método não é importante, pode ser qualquer um. O importante é que você recebe um objeto do tipo `WebDataBinder` e, com ele, é possível registrar novos validators. A linha `binder.setValidator(new ProductValidator());` mostra exatamente essa opção. Através desse objeto, também podemos adicionar novos conversores de tipos e informar quais campos podem ser enviados pelo formulário, entre outras coisas.

Para esse código já ficar funcional, precisamos adicionar as dependências necessárias para podermos usar a Bean Validation. No nosso caso, vamos usar o Hibernate Validator como escolha de implementação dessa especificação.

```
<!-- JSR 303 with Hibernate Validator -->
<dependency>
    <groupId>javax.validation</groupId>
    <artifactId>validation-api</artifactId>
    <version>1.0.0.GA</version>
</dependency>
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>4.1.0.Final</version>
</dependency>
```

Uma última informação relevante: adicionamos o `ProductValidator` como sendo o validador que será utilizado para qualquer formulário que envolva o cadastro de novos livros. O ponto aqui é: podem existir situações em que o formulário seja um pouco diferente e, aí, não vamos ter uma associação tão direta com nossa classe de modelo. Nessas ocasiões, uma estratégia é criar uma outra classe que represente justamente esse formulário.

```
public class UpdateProductForm {
    // atributos que podem não existir na classe Product.
}
```

Poderia ser criado um validator específico para este formulário.

```
public class UpdateProductFormValidator implements Validator{
    ....
}
```

Por fim, você adicionaria mais esse validador no `WebDataBinder`, anotaria o parâmetro do método com `@Valid` e o Spring MVC poderia utilizá-lo. Só que a pergunta é: como o framework vai saber que para determinado parâmetro ele deve usar um validador e que, para outro parâmetro, ele deve usar outro? A resposta é o método `supports`, que também deve ser implementado dentro da classe que implementa a interface `Validator`.

```
public class ProductValidator implements Validator{

    @Override
    public boolean supports(Class<?> clazz) {
        return Product.class.isAssignableFrom(clazz);
    }

    ...
}
```

Esse método recebe a classe do objeto que está querendo ser validado e retorna se o validador consegue lidar com ele. Essa é a forma como o Spring MVC controla qual validação deve ser aplicada. Inclusive, é uma ótima maneira de separar regras de validação em várias classes e deixar que o Spring MVC execute cada uma delas para você.

Uma curiosidade sobre o uso das validações customizadas do Spring: caso você associe uma classe de validação a um controller, mas seu método `supports` não aceite o parâmetro anotado com `@Valid`, será lançada uma exception parecida com a que segue.

```
java.lang.IllegalStateException: Invalid target for Validator
br.com.casadocodigo.loja.validacao.ProductValidator
```

Voltando para a tela de erro

Com a validação básica configurada, não é mais permitido que façamos o cadastro de um produto com dados inválidos. O problema é que a tela de erro, neste momento, não é muito amigável.

Fig. 5.1: Status de request inválido

O código de erro 400 indica que a requisição foi executada com dados inválidos. Entretanto, nosso usuário não entende nada disso, o que ele quer é voltar para a tela de formulário, para preencher os campos devidamente.

```
@RequestMapping(method=RequestMethod.POST)
public String save(@Valid Product product,
    BindingResult bindingResult,
    RedirectAttributes redirectAttributes){
    if(bindingResult.hasErrors()){
        return "produtos/form";
    }
    productDAO.save(product);
    redirectAttributes.addFlashAttribute("sucesso",
      "Produto cadastrado com sucesso");
    return "redirect:produtos";
}
```

Para conseguirmos verificar se o formulário teve erro ou não, fazemos uso do objeto do tipo `BindingResult`. Essa classe possui vários métodos, tanto para verificar se existem erros de validação, quanto para acrescentar novos erros, caso o programador queira efetuar alguma lógica de validação diretamente no método do controller.

Reaproveitando métodos no controller

Ainda precisamos completar nosso fluxo, já que os dados enviados pelo usuário não estão sendo mostrados no formulário, após a validação. Além disso, ainda temos outro problema, os campos de preços por tipo de livro não estão mais sendo exibidos. O problema é que a gente retorna o usuário para o formulário, mas não adiciona no `ModelAndView` os tipos de livros. Para resolver isso, podemos fazer uma leve modificação no método `save`, para que seja possível reaproveitar a lógica do método `form`.

```
@RequestMapping(method=RequestMethod.POST)
public ModelAndView save(@Valid Product product,
    BindingResult bindingResult,
    RedirectAttributes redirectAttributes){
    if(bindingResult.hasErrors()){
        return form();
    }
    productDAO.save(product);
    redirectAttributes.addFlashAttribute("sucesso",
        "Produto cadastrado com sucesso");
    return new ModelAndView("redirect:produtos");
}
```

Agora, caso seja encontrado algum erro de validação, é retornado o objeto do tipo `ModelAndView` criado pelo método `form`, que já é responsável por disponibilizar os tipos de livro na view. Caso tudo tenha ocorrido bem, é retornado um `ModelAndView` com a String de redirect para URL de listagem de produtos.

5.2 Exibindo os erros

Nosso usuário, após um problema de validação, já é retornado para o formulário inicial. Só que ainda não mostramos os erros e nem estamos mantendo os dados preenchidos por ele. Esses são dois problemas que precisam ser resolvidos.

Vamos começar pela exibição dos erros na página. Sempre que algum erro acontece, o Spring MVC disponibiliza um objeto do tipo `BindingResult` para a JSP, contendo todas as informações necessárias. Lembre-se que esse objeto é o mesmo que usamos no método do controller para verificar se foram encontrados erros de validação.

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

...

<c:forEach
    items="${requestScope['org.springframework.validation.
        BindingResult.product'].allErrors}" var="error">

    ${error.code}<br/>

</c:forEach>
```

O método `allErrors` retorna uma lista de objetos do tipo `FieldError`, o qual possui as informações relevantes sobre cada problema encontrado no processo de validação.

Essa maneira de exibir os errors até funciona, o único problema dela é o nome horrível da variável. Isso é uma decisão interna do framework e, em geral, esse tipo de coisa pode mudar facilmente, até de um update para outro. Dessa forma, não é bom basear qualquer código da nossa aplicação em uma String como essa.

Usando a tag hasBindErrors

Para facilitar a exibição das mesmas mensagens de erro, podemos usar uma *taglib* fornecida pelo Spring MVC.

```
<%@ taglib prefix="spring"
 uri="http://www.springframework.org/tags"%>
...

<spring:hasBindErrors name="product">
    <ul>
    <c:forEach var="error" items="${errors.allErrors}">
        <li>${error.code}</li>
    </c:forEach>
    </ul>
</spring:hasBindErrors>
```

A tag `hasBindErrors` disponibiliza o mesmo objeto do tipo `BindingResult` para você, só que agora sob a variável `errors`. A vantagem é que o código não fica acoplado com uma String criada dentro do framework. Ela ainda possui o atributo `name` que, curiosamente, recebeu como argumento a String `produto`. Todos os erros de validação ou conversão, capturados pelo processo Spring MVC, são associados a algum parâmetro em questão.

```
public String save(@Valid Product product,...){
    ...
}
```

No caso do método `save` o tipo do parâmetro é `Product` e, exatamente por isso, passamos uma String de mesmo nome, com a primeira letra em minúsculo, para o atributo `name`, da tag. Isso é muito importante, pois caso o método possua outros parâmetros que devam ser validados, os erros de cada um ficarão disponíveis associados ao tipo do mesmo parâmetro. Para ficar ainda mais claro, vamos lembrar do primeiro código utilizado para exibir os problemas de validação.

```
<c:forEach
    items="${requestScope['org.springframework.validation.
        BindingResult.product'].allErrors}" var="error">
    ${error.code}<br/>

</c:forEach>
```

Perceba que a variável acaba com `.product` e é justamente essa mesma variável que será usada internamente pela tag. Para o leitor mais curioso, segue o trecho de código da tag do Spring MVC responsável por isso.

```
public class RequestContext {
    ....

    public Errors getErrors(String name, boolean htmlEscape) {
        Errors errors = this.errorsMap.get(name);
        boolean put = false;
        if (errors == null) {
            errors = (Errors) getModelObject(
                BindingResult.MODEL_KEY_PREFIX + name);
            // Check old BindException prefix for backwards
            // compatibility.
            if (errors instanceof BindException) {
                errors =
                    ((BindException) errors).getBindingResult();
            }

            ....
    }
}
```

A tag `hasBindErrors` faz parte do conjunto de tags sob a URI http://www.springframework.org/tags. Esse mesmo namespace ainda possui diversas outras tags que podem ser úteis para sua aplicação. Alguns exemplos:

- `message`: uma extensão da tag de mensagens padrão, que existe na JSTL.
- `transform`: para aplicar conversões de tipos específicos para String.
- `url`: uma extensão da tag URL padrão, que existe na JSTL
- `mvcUrl`: função que ajuda a referenciar as URLs dos métodos dentro da JSP.

A função `mvcUrl` merece uma atenção especial, pois vai nos ajudar a contornar um problema que já estamos enfrentando. Até agora, na action do

formulário de cadastro, para referenciarmos o endereço que leva até o método de cadastro, foi usada a URL completa.

```
/casadocodigo/produtos
```

Isso até funciona, mas temos dois problemas:

- caso o nome do contexto mude, temos que mudar a URL;
- caso a URL mude, seremos obrigados a mudar em todos lugares.

Para resolver os dois casos, existe a função `mvcUrl`.

```
<form action="${spring:mvcUrl("PC#save").build()}" method="post">
```

Essa função recebe como parâmetro as letras em maiúsculo do seu controller e o nome do método, separados pelo `#`. O método `build()` serve para realizar a construção da URL. A saída ficaria assim:

```
http://localhost:8080/casadocodigo/produtos
```

A vantagem é que, caso você queira mudar a URL do seu controller, você não precisará mudar os links da sua página. Para finalizar, se você não gostar da sintaxe com o `#`, é possível usar o atributo `name` da annotation `@RequestMapping` e definir um nome para ser usado dentro da função.

```
@RequestMapping(method=RequestMethod.POST,name="saveProduct")
public ModelAndView save(....){
    ....
}
```

Agora, quando a função **mvcUrl** for utilizada, ela pode receber como argumento o valor passado para o atributo *name* da annotation.

```
${spring:mvcUrl("saveProduct").build()}
```

5.3 Exibindo as mensagens de erro de maneira amigável

Por mais que o formulário esteja sendo validado, as mensagens de erro ainda não dizem muita coisa sobre o problema em questão. Atualmente, o máximo que podemos conseguir é exibir os erros da seguinte forma:

```
numeroPaginas - typeMismatch
titulo - field.required
descricao - field.required
numeroPaginas - field.required
```

Apenas para não ficar dúvidas, a mensagem `typeMismatch` indica que aconteceu um problema de conversão. Nesse caso o motivo é que o campo número de páginas foi deixado em branco e no modelo `Product`, ele é um atributo do tipo *int*.

Para resolver esse problema precisamos ensinar ao Spring MVC onde ele deve buscar as mensagens corretas e, para isso, vamos criar um arquivo externo contendo as mesmas. Vamos dar uma olhada no conteúdo dele.

```
field.required = Campo obrigatorio
field.required.product.title = Titulo obrigatorio
```

Aqui configuramos uma mensagem padrão para a chave `field.required` e ainda optamos por deixar uma específica para o título do produto em questão, `field.required.product.title`

Esse arquivo deve ser criado dentro da sua pasta `WEB-INF` e geralmente é chamado de `messages.properties`. Agora é necessário avisar ao Spring MVC que ele deve usá-lo para buscar as mensagens relativas às chaves que estão configuradas. Para isso, vamos adicionar um novo método na classe `AppWebConfiguration`.

```
@Bean
public MessageSource messageSource(){
    ReloadableResourceBundleMessageSource bundle =
        new ReloadableResourceBundleMessageSource();
    bundle.setBasename("/WEB-INF/messages");
    bundle.setDefaultEncoding("UTF-8");
```

```
    bundle.setCacheSeconds(1);
    return bundle;
}
```

Perceba que indicamos a localização do arquivo base de mensagens. Mais para a frente, vamos voltar a utilizá-lo para fazer nossa aplicação suportar vários idiomas. Um método interessante é o `setCacheSeconds`. Ele serve para indicar que o arquivo de mensagens deve ser recarregado a cada intervalo de tempo, no caso, foi configurado apenas um segundo. Essa é uma configuração típica de ambiente de desenvolvimento, já que não queremos ficar parando e subindo o servidor a cada alteração desse arquivo.

Um detalhe muito importante, mas não muito claro, é que o nome do método deve ser `messageSource`. O Spring MVC vai procurar por um Bean registrado com esse nome, para que possa carregar as mensagens. Uma alternativa, para não ter que se preocupar com o nome do método, é utilizar o atributo *name* da annotation `@Bean`.

```
@Bean(name="messageSource")
public MessageSource loadBundle(){
    ...
}
```

Na opinião deste autor não existe um melhor que o outro. Até porque você não fica alterando o nome dos métodos de configuração. Não perca tempo com isso, apenas faça a configuração necessária para que você possa continuar o desenvolvimento da da aplicação.

Agora, finalmente, estamos prontos para exibir as mensagens dentro da nossa página.

```
<spring:hasBindErrors name="product">
    <ul>
        <c:forEach var="error" items="${errors.allErrors}">
            <li><spring:message code="${error.code}"
                text="${error.defaultMessage}"/></li>
        </c:forEach>
    </ul>
</spring:hasBindErrors>
```

A tag message, como mostramos, serve para buscar as mensagens associadas às chaves. O atributo text nos ajuda a deixar uma mensagem padrão onde as chaves ainda não estiverem configuradas no arquivo.

Alterando a mensagem de conversão

Voltando um pouco para os problemas de validação, um outro erro que pode existir no formulário de cadastro é a entrada de um valor não condizente com o tipo do campo. Por exemplo, um usuário pode digitar "dez" em vez de "10" no campo de número de páginas. Esse tipo de situação também acontece, em geral, com campos de datas. Para exibir mensagens amigáveis, relativas aos erros de conversão, adicionamos as chaves que começam com typeMismatch.

```
field.required = Campo obrigatorio
field.required.product.title = Titulo obrigatorio
typeMismatch.product.pages = Digite um número, exemplo "100"
typeMismatch.java.lang.Integer = Digite apenas números
typeMismatch.int = Digite apenas numeros
typeMismatch = O tipo esta errado
```

Perceba que adicionamos as mensagens para problemas de conversão de tipos inteiros no geral e ainda indicamos uma mais específica para o atributo pages.

Posicionamento da mensagem de erro

As mensagens de erros estão sendo exibidas no início da nossa tela, mas não é incomum que o cliente requisite que a mesma mensagem seja exibida ao lado do campo. E é justamente o que vamos fazer agora.

```
<spring:hasBindErrors name="product">

    ...

    <form action="${spring:mvcUrl("PC#save").build()}"
        method="post">
```

```
        <div>
            <label for="title">Titulo</label>
            <input type="text" name="title"
                id="title" />
            <c:forEach var="error"
                items="${errors.getFieldErrors('title')}">
               <span>
                   <spring:message code="${error.code}"
                    text="${error.defaultMessage}"/>
               </span>
            </c:forEach>
        </div>

    </form>
</spring:hasBindErrors>
```

Fizemos uma pequena modificação e deixamos o formulário inteiro dentro da tag `hasBinderrors`. Com essa alteração, podemos usar a variável `errors` do lado de cada campo dele e neste caso foi usado o método `getFieldErrors` para pegar o erro de cada atributo.

O único problema desse código é que ele provavelmente será necessário em várias páginas do nosso sistema. Para não termos que ficar copiando e colando esse código, vamos utilizar uma outra tag provida pelo Spring MVC.

```
<%@ taglib prefix="spring"
    uri="http://www.springframework.org/tags"%>
<%@ taglib prefix="form"
    uri="http://www.springframework.org/tags/form"%>

<form:form action="${spring:mvcUrl("PC#save").build()"
    method="post"
    commandName="product">

    <div>
        <label for="title">Titulo</label>
        <input type="text" name="title" id="title"/>
        <form:errors path="title"/>
    </div>
```

```
</form:form>
```

O código ficou muito mais legível! A tag `errors` faz praticamente o mesmo papel do nosso código anterior. Ela vai exibir todos os erros do atributo em questão. Ela ainda possui alguns atributos para que você possa definir, entre outras coisas, o estilo e a tag HTML que vai ser utilizada para renderizar os erros. Para que a tag `errors` funcione, é necessário que ela seja usada dentro da tag `form`.

```
<form:form action="${spring:mvcUrl("PC#save").build()"
    method="post"
    commandName="product">
        ...
        <form:errors path="title"/>
</form:form>
```

O ponto que precisa de mais atenção na tag `form` é o atributo `commandName`. Ele tem o mesmo propósito do atributo `name` na tag `hasBindErrors`, ou seja, recebe o tipo do parâmetro que está sendo validado pelo nosso controller, com a primeira letra em minúsculo. Apenas para refrescar a memória:

```
save(@Valid Product product....)
```

Um último detalhe, foi necessário importar mais um conjunto de tags para a nossa página.

```
<%@ taglib uri="http://www.springframework.org/tags/form"
    prefix="form"%>
```

Esse conjunto possui tags que nos ajudam a escrever um formulário em si. Seguem alguns exemplos:

- checkbox
- select
- hidden

- form

Na próxima seção já vamos fazer um pouco mais de uso dessas tags.

5.4 Mantendo os valores no formulário

Para completar nosso formulário, precisamos dar a possibilidade de o usuário continuar o preenchimento de onde ele parou. Atualmente, caso aconteça algum erro de validação, voltamos para a mesma tela, mas não mantemos as informações previamente preenchidas. Vamos ver como o Spring MVC pode nos ajudar.

```
<form:form action="${spring:mvcUrl("PC#save").build()"
    method="post" commandName="product">
    <div>
        <label for="title">Titulo</label>
        <form:input path="title"/>
        <form:errors path="title"/>
    </div>
    <div>
        <label for="description">Descrição</label>
        <form:textarea path="description" rows="10"
        cols="20"/>
    </div>
    ...
    <div>
        <label for="releaseDate">Data de lançamento</label>
        <form:input path="releaseDate" type="date"/>
        <form:errors path="releaseDate"/>
    </div>
</form:form>
```

Fizemos uso daquele conjunto de tags específicas para o formulário. Além da `errors`, usamos a `input` e a `textarea`. Perceba que todas têm o atributo `path`, que define a propriedade do modelo que queremos usar. Vale lembrar que não estamos nos referindo ao atributo privado e sim aos `getters` expostos na classe. Basicamente, o que essas tags fazem é acessar

a variável com o mesmo nome do `commandName`, e ir invocando os getters necessários.

O método form e o atributo commandName

Caso o leitor mais curioso tenha executado o código com as tags de ajuda do formulário, neste momento, ele terá recebido uma exception um tanto quanto estranha.

```
java.lang.IllegalStateException: Neither BindingResult nor
plain target object for bean name 'product' available as
request attribute
```

Quando acessamos a tela do formulário pela primeira vez, através da URL `/produtos/form`, fazemos uso do método `form()`, da classe `ProductsController`. Já foi explicado que, para a tag `form:form` funcionar, ela precisa de acesso ao objeto recebido como parâmetro. No caso desse nosso método, onde está o parâmetro? É justamente por esse motivo que estamos recebendo essa exception.

Para conseguirmos contornar isso, vamos fazer uma leve alteração.

```
public ModelAndView form(Product product) {
    ...
}
```

Forçamos o método `form` a receber um parâmetro do tipo `Product`, apenas para o Spring MVC disponibilizá-lo como variável do request. Caso você esteja pensando que essa alteração está sendo imposta pelo framework, vá se acostumando. Muitas vezes temos que escrever um código que não queremos apenas para agradar a tecnologia em questão, mas dado que já ganhamos bastante, é um tradeoff mais do que justo. Agora, para nosso código voltar a compilar, é necessário alterar a invocação do método `form` a partir do método `save`.

```
public ModelAndView cadastra(@Valid Product product...){
    if(bindingResult.hasErrors()){
        return form(product);
    }
}
```

A annotation @ModelAttribute

Estamos sempre configurando o atributo `commandName` com o mesmo nome do tipo do parâmetro recebido no método. Caso você queira alterar isso, por exemplo passando o valor "objetoAtual", pode utilizar a annotation `@ModelAttribute`.

```
save(@Valid @ModelAttribute("objetoAtual") Product product)
```

Essa annotation pode ser útil caso, por algum motivo, sua empresa tenha uma padrão para nomes de variáveis disponíveis na sua JSP.

5.5 Integração com a Bean Validation

A nossa validação já funciona integralmente, mas ainda tem um ponto que, pelo menos para este autor, ainda é um pouco ruim. O nosso validador customizado realiza verificações muito básicas.

```
ValidationUtils.rejectIfEmptyOrWhitespace(errors,
    "title", "field.required");
```

Esse tipo de validação é tão comum que, como já comentamos, virou até especificação no mundo Java, a *Bean Validation*. E como não podia ser diferente, o módulo de validação é completamente integrado com ela. Só para lembrar o leitor, já configuramos o Maven para utilizarmos a implementação feita pelo time do Hibernate, chamada de *Hibernate Validator*. O primeiro passo, para a felicidade de qualquer equipe de programação, é que não precisamos do nosso validador customizado. Logo, lembre de apagá-lo do seu código, ou pelo menos não configure mais para ele ser utilizado pelo Spring MVC.

```
//por enquanto não precisamos mais desse metodo.
//Comente ou apague.
@InitBinder
protected void initBinder(WebDataBinder binder) {
    binder.setValidator(new ProdutoValidator());
}
```

Agora precisamos de outra forma para ensinar o Spring MVC que ele precisa validar os dados básicos de um livro.

```
@Entity
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    @NotBlank
    private String title;
    @Lob
    @NotBlank
    private String description;
    @Min(30)
    private int pages;
```

Pronto, com algumas annotations conseguimos o mesmo efeito de validação. Agora, caso você teste um novo cadastro inválido, as mensagens de validação ainda não estarão da maneira como esperamos.

- @NotBlank -> "may not be empty"
- @Min -> "must be greater than or equal to ?"

Como já fizemos antes, vamos alterar o arquivo `message.properties` e adicionar as novas chaves de validação.

```
NotBlank.java.lang.String = Campo obrigatorio
NotBlank.product.title = Titulo obrigatório
Min.product.pages = O numero minimo de paginas é {1}
```

Perceba que simplesmente adicionamos o nome da annotation na frente da chave, o resto segue o mesmo padrão que já utilizamos. Um ponto a ser observado é a mensagem referente ao mínimo de páginas. Temos que usar o valor do argumento que foi configurado na annotation.

```
@Min(30)
private int pages;
```

O Spring MVC disponibiliza isso como parâmetro da sua mensagem. No caso da mensagem de validação, o argumento `{0}` sempre é o nome do atributo sendo validado. Por isso, foi usado o `{1}`. Caso você queira alterar as mensagens padrões, que estão associadas a cada uma das annotations da Bean Validation, siga este link: http://tinyurl.com/oocl5ro.

```
...
org.hibernate.validator.constraints.NotBlank.message =
    may not be empty
```

Ele vai lhe fornecer as chaves necessárias, que devem ser substituídas, para que o Hibernate Validator trabalhe com as mensagens customizadas. Basta que você copie no seu arquivo `messages` e troque o valor associado a ela.

```
...
org.hibernate.validator.constraints.NotBlank.message =
    Campo obrigatório
```

5.6 Convertendo a data

Na Casa do Código, muitos livros são escritos o tempo todo. Para ter um fluxo interessante de lançamentos, foi decidido que os livros cadastrados devem ter uma data de lançamento.

```
@Entity
public class Product {
    ...
    private Calendar releaseDate;

    //getter e setter.
}
```

Agora precisamos colocar um campo novo no nosso formulário.

```
<div>
    <label for="releaseDate">Data de lançamento</label>
    <input type="date" name="releaseDate"/>
    <form:errors path="releaseDate"/>
</div>
```

Caso nosso usuário efetue o teste, aparecerá uma mensagem de falha de conversão.

```
Failed to convert property value of type java.lang.String to
required type java.util.Calendar for property releaseDate;
nested exception is java.lang.IllegalStateException: Cannot
convert value of type [java.lang.String] to required type
[java.util.Calendar] for property releaseDate: no matching
editors or conversion strategy found
```

A exception indica que o Spring MVC não conseguiu achar, de maneira automática, um conversor de String para o tipo `Calendar`. Para que este problema seja resolvido, é necessário usarmos uma annotation indicando que queremos a conversão.

```
@DateTimeFormat
private Calendar releaseDate;
```

Agora, se tentarmos mais uma vez o cadastro, receberemos a seguinte exception de conversão:

```
Failed to convert property value of type java.lang.String to
required type java.util.Calendar for property releaseDate;
nested exception is
org.springframework.core.convert.ConversionFailedException:
Failed to convert from type java.lang.String to type
@org.springframework.format.annotation.DateTimeFormat
java.util.Calendar for value 2015-02-20; nested exception is
java.lang.IllegalArgumentException: Unable to parse 2015-02-20
```

O problema é que o formato default usado pelo conversor registrado no Spring MVC é o `M/d/yy`, representado pela constante `DateFormat.SHORT`; e o valor enviado pelo nosso formulário, já que usamos o input cujo tipo é o `date`, sempre vai no formato `yyyy-MM-dd`. Para a nossa sorte, esse formato é o muito usado no mundo http://pt.wikipedia.org/wiki/ISO_8601 e o Spring MVC já tem o suporte pronto.

```
@DateTimeFormat(iso=ISO.DATE)
private Calendar releaseDate;
```

Redefinindo o formato global

Sempre que temos de lembrar de fazer alguma configuração no nosso código, por vários motivos, acabamos esquecendo e passando muito tempo tentando corrigir. É justamente o que pode acontecer com a nossa estratégia de conversão atual. Sempre temos que nos lembrar de pôr o estilo de conversão na annotation `@DateTimeFormat`.

Para melhorar essa parte da nossa aplicação, vamos ensinar ao Spring MVC que ele sempre deve usar essa formatação. Vamos adicionar mais um método na classe `AppWebConfiguration`.

```
@Bean
public FormattingConversionService mvcConversionService() {
    DefaultFormattingConversionService conversionService =
        new DefaultFormattingConversionService(true);

    DateFormatterRegistrar registrar =
        new DateFormatterRegistrar();
    registrar.setFormatter(new DateFormatter("yyyy-MM-dd"));
    registrar.registerFormatters(conversionService);
    return conversionService;
}
```

Respire, pois está acontecendo muita coisa nesse código. A primeira informação é que o nome do método **tem** que ser `mvcConversionService`. Pois esse é o nome usado internamente pelo Spring MVC para registrar o objeto responsável por agrupar os conversores. Para que isso não pareça tão mágico, o leitor pode checar a classe `WebMvcConfigurationSupport`, diretamente no código-fonte do Spring, http://tinyurl.com/na3g9jp.

O próximo passo importante é o uso do objeto do tipo `DateFormatterRegistrar`. Ele implementa a interface `FormatterRegistrar`, que deve ser usada quando é necessário agrupar vários tipos de conversões. No caso da data, temos várias possibilidades, seguem alguns exemplos.

- Calendar para Long
- Long para Calendar

- Date para Calendar
- Calendar para Date

Depois de criarmos o objeto do tipo `DateFormatterRegistrar`, informamos qual a formatação que deve ser aplicada por padrão.

```
registrar.setFormatter(new DateFormatter("yyyy-MM-dd"));
```

Por fim, é feito o registro de todos os conversores no objeto do tipo `FormattingConversionService`. O mesmo é instanciado com um `true` no construtor, indicando que todos conversores padrões, devem ser adicionados.

Entendendo mais sobre os formatadores do Spring MVC

Uma das belezas dos formatadores do Spring MVC é que a nossa configuração de formatação das datas vale para os dois lados da moeda.

- A String de entrada é convertida para o `Calendar` do seu modelo.
- O `Calendar` do seu modelo é formatado para ser apresentado no formulário.

Para ficar mais claro, dê uma olhada na interface que define os formatadores.

```
public interface Formatter<T> extends Printer<T>, Parser<T> {
}
```

As tags do Spring MVC tiram proveito dos formatadores, com isso não precisamos ficar nos preocupando em tratar os dados que devem ser exibidos nos inputs dos formulários.

```
<div>
    <label for="releaseDate">Data de lançamento</label>
    <form:input path="releaseDate" type="date"/>
    <form:errors path="releaseDate"/>
</div>
```

Perceba que só indicamos o nome da propriedade que deve ser usada e, pelo tipo da mesma, o Spring MVC vai decidir qual formatador utilizar.

5.7 Conclusão

Conseguimos cobrir tudo que é necessário para você aplicar as validações necessárias nos sistemas que serão desenvolvidos. Passamos pela fase onde você mesmo poderia criar o seu validador e vimos como lidar com o erro de validação redirecionando o usuário para o local correto. Também analisamos as possibilidades de configuração das mensagens de erros e como exibir as mesmas dentro das páginas.

Para fechar, fomos além e vimos a integração com as annotations da Bean-Validaton e entendemos como funcionam os conversores de tipos dentro do Spring MVC. Minha dica, nesse momento, é que você descanse um pouco, tente praticar o que vimos até aqui e só depois continue a leitura.

Capítulo 6

Upload de arquivos

Uma característica importante das livrarias online é a disponibilização do sumário dos livros que estão à venda. A Casa do Código, como não poderia deixar de ser, também deve oferecer essa possibilidade a seus usuários.

6.1 Recebendo o arquivo no Controller

Vamos começar alterando a página da Casa do Código responsável pela entrada de dados de cada um dos novos livros.

```
<form:form action="${spring:mvcUrl('PC#save').build()}"
    method="post"
    commandName="product">

    ...
```

```
<div>
    <label for="summary">Sumario do livro</label>
    <input type="file" name="summary"/>
    <form:errors path="summaryPath"/>
</div>

...
```

A única mudança foi a adição do input que vai receber o arquivo que, neste caso, deve ser do tipo `file`. Agora, se tentarmos enviar os dados desse formulário, devemos receber o seguinte resultado:
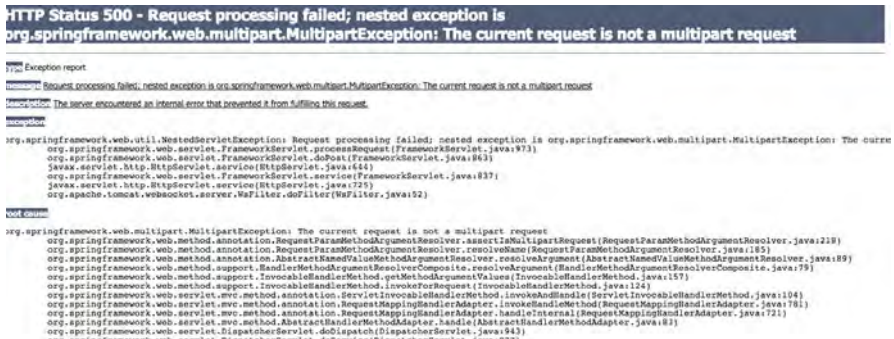
Fig. 6.1: O request não é multipart

"The current request is not a multipart request

signifca que tentamos enviar o arquivo de uma maneira que o Spring MVC não consegue entender. Basicamente, o que acontece é que todos os dados do formulário são enviados de alguma forma e, o default é que eles sejam submetidos usando o formato `application/x-www-form-urlencoded`. Desse modo, os dados são enviados seguindo quase que as mesmas regras utilizadas quando utilizamos o método `GET`. Uma sequência de chaves e valores.

A nossa situação atual exige que façamos uma alteração nesse formato, de modo que o arquivo possa ser integralmente passado do navegador para o servidor. É por isso que vamos utilizar o formato `multipart/form-data`.

```
<form:form action="${spring:mvcUrl('PC#save').build()}"
    method="post"
    commandName="product"
    enctype="multipart/form-data">


    ...
</form>
```

O atributo `enctype` serve justamente para indicarmos como queremos mandar os dados do navegador para o servidor e o `multipart/form-data` é o jeito utilizado quando necessitamos transmitir arquivos. Por exemplo, já existe um trabalho sendo feito para que o seu *form* possa enviar os dados através do formato `JSON`. Caso fique curioso, navegue até http://www.w3.org/TR/html-json-forms/.

Alterando o método do controller

Agora que nosso formulário já envia o arquivo, precisamos recebê-lo e lidar com ele. Faremos isso no método `save` da classe `ProductsController`.

```
public class ProductsController {
    @RequestMapping(method=RequestMethod.POST)
    public ModelAndView save(
    Part summary,@Valid Product product,...) {
        System.out.println(summary.getName() + ";"
            +summary.getHeader("content-disposition"));
        //resto do código
    }
}
```

A parte importante desse código é o parâmetro do tipo `Part`. Ele foi introduzido a partir da versão 3 da especificação de Servlets e a interface responsável por representar o arquivo que foi enviado pelo formulário. Perceba que o nome do argumento é o mesmo nome do input que declaramos no formulário. Aqui usamos dois métodos:

- `getName()`: retorna o nome do input usado no formulário;

- `getHeader()`: usado para recuperar alguma informação sobre o que foi enviado.

E caso a gente queira recuperar o nome do arquivo, como vamos fazer? O leitor mais experiente deve estar pensando que tem algum método que já retorna isso pronto e esse é justamente o problema. A interface `Part` é a maneira mais primitiva de tratarmos os dados enviados através do *enctype* `multipart/form- data`. Na verdade, podemos acessar o valor de cada um dos campos do formulário, basta que declaremos um parâmetro do tipo `Part` para cada input. Para ficar um pouco mais claro, dê uma olhada na maneira como esses dados são enviados a partir do navegador.

Fig. 6.2: Corpo do request para uma requisição multipart

Para os campos que não representam arquivos, o Spring MVC já nos ajuda bastante e faz a ligação direta com as propriedades do nosso modelo. Para não nos decepcionar, ele também oferece um jeito mais fácil de lidar com o arquivo que foi submetido.

```
public class ProductsController {
    @RequestMapping(method=RequestMethod.POST)
    public ModelAndView save(MultipartFile summary,
        @Valid Product product,...) {
```

```
        System.out.println(summary.getName() + ";"
            +summary.getOriginalFilename());
        //resto do código
    }
}
```

O método `getName` continua fazendo a mesma coisa, só que agora já temos outro método, o `getOriginalFilename()`, que já nos retorna o nome do arquivo que foi enviado.

6.2 Salvando o caminho do arquivo

Agora que já temos o objeto que representa o arquivo enviado em nossas mãos, é necessário gravá-lo em algum lugar.

```
public class ProductsController {

    @Autowired
    private FileSaver fileSaver;

    @RequestMapping(method=RequestMethod.POST)
    public ModelAndView save(MultipartFile summary,
        @Valid Product product,...) {
        //faz a validação

        String webPath =
            fileSaver.write("uploaded-images",summary);
        product.setSummaryPath(webPath);
        products.save(product);

        //fim do método
    }
}

@Component
public class FileSaver {

    @Autowired
    private HttpServletRequest request;
```

```
    public String write(String baseFolder, MultipartFile file) {
        String realPath = request.getServletContext()
            .getRealPath("/"+baseFolder);
        try {
            String path =
                realPath+"/"+file.getOriginalFilename();
            file.transferTo(new File(path));
            return baseFolder+"/"+file.getOriginalFilename();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }

    }
}
```

Dê uma respirada e olhe com cuidado para o código. Perceba que não fizemos nada demais, apenas criamos outra classe responsável por gravar o arquivo em uma pasta da nossa aplicação. A ideia é não deixar que esse código atrapalhe o fluxo do método `save`, no nosso controller. Além disso é necessário criar o atributo `summaryPath` na classe **Product**, juntamente com o seu getter e setter.

Dessa forma, caso seja necessário criar um link para o sumário, basta que façamos algo parecido com o que segue:

```
<img src="${product.summaryPath}"/>
```

6.3 Configurações necessárias

Caso você tenha rodado a aplicação, percebeu que nosso upload de arquivos ainda não está funcionando. Está sendo lançada uma exception do tipo `IllegalArgumentException`.

```
java.lang.IllegalArgumentException: Expected
MultipartHttpServletRequest: is a MultipartResolver configured?
```

A interface `MultipartResolver` é a que define os métodos necessários para o tratamento inicial de um request cujo modo de envio, também

conhecido como `contentType`, é o `multipart/form-data`. Precisamos prover uma implementação dessa interface para que o Spring MVC possa fazer seu trabalho. Para isso, basta que adicionemos mais um método na classe `AppWebConfiguration`.

```
@EnableWebMvc
@ComponentScan(basePackageClasses={HomeController.class,
    ProductDAO.class,FileSaver.class})
public class AppWebConfiguration {
    ...
}

@Bean
public MultipartResolver multipartResolver(){
    return new StandardServletMultipartResolver();
}
```

Aqui foi escolhida a implementação que usa o recurso de tratamento de upload provido pela especificação de Servlets. Outra alternativa é usar a implementação baseada na biblioteca `commons-fileupload`. Nesse caso, a classe escolhida seria a `CommonsMultipartResolver`.

Além disso, na annotation `@ComponentScan`, passamos a classe `FileSaver` com o objetivo de que o Spring vasculhe o pacote em que ela se encontra.

Agora, caso o leitor tente salvar um novo livro, vai receber mais uma exception. Só que desta vez ela é diferente, o que indica que resolvemos o problema anterior.

```
java.lang.IllegalStateException: Unable to process parts as
    no multi-part configuration has been provided
```

A parte mais importante dessa exception é: *"no multi-part configuration has been provided*

. Quando configuramos o tratamento de upload, podemos definir algumas coisas:

- Local de armazenamento temporário enquanto o arquivo está sendo recebido;

- Tamanho máximo do arquivo;
- Tamanho máximo do request como um todo.

Na especificação de Servlets, essa configuração é feita através da classe `MultipartConfigElement`.

```
//outros imports
import javax.servlet.ServletRegistration.Dynamic;

public class ServletSpringMVC extends
    AbstractAnnotationConfigDispatcherServletInitializer {

    ...

    @Override
    protected void customizeRegistration(Dynamic registration) {
        registration.setMultipartConfig(
            new MultipartConfigElement(""));
    }
}
```

Só para lembrar, a classe `ServletSpringMVC` fornece itens básicos como: mapeamento de URL do servlet do Spring MVC e quais outras classes de configurações devem ser carregadas. Agora, além disso, sobrescrevemos o método `customizeRegistration`. Ele recebe como parâmetro um objeto do tipo `Dynamic`, que nos possibilita, entre outras coisas, registrar o nosso objeto de configuração do tipo `MultipartConfigElement`. O construtor recebendo apenas uma String vazia indica que o próprio servidor web vai decidir qual é o local de armazenamento temporário dos arquivos.

Pronto, agora nossa aplicação está pronta para receber qualquer tipo de arquivo através dos formulários.

6.4 Gravando os arquivos fora do servidor web

Nossa lógica de upload grava os novos arquivos em uma pasta dentro da própria aplicação web. O problema dessa abordagem, mesmo quando estamos

em ambiente de desenvolvimento, é que a cada nova alteração que temos no projeto a nossa IDE força um *hot deploy* no servidor. Em geral, o *hot deploy* destrói a aplicação que estava no servidor e cria uma nova, fazendo com que percamos os arquivos que já tinham sido enviados.

Além disso, um fato que incomoda o desenvolvedor é perceber que os arquivos enviados não ficam exatamente dentro do projeto que ele está visualizando, por exemplo, no Eclipse e sim na instalação do servidor que foi escolhida. Para simplificar tudo isso, facilitando inclusive o processo de deploy que discutiremos mais para o fim do livro, é que podemos usar um serviço como o *Amazon S3*. Basicamente, a Amazon fornece um servidor onde podemos enviar nossos arquivos e eles ficam disponíveis para serem acessados pela web. Eles possuem servidores dedicados só para isso.

Integração com o Amazon S3

Para quem estiver interessado nessa opção, a primeira coisa que devemos fazer é alterar o código que está na classe `FileSaver`, que é a responsável por gravar o arquivo.

```
@Component
public class FileSaver {

    public String write(String baseFolder,
    MultipartFile multipartFile) {
        AmazonS3Client s3 = client();
        try {
            s3.putObject("casadocodigo",
                multipartFile.getOriginalFilename(),
                    multipartFile.getInputStream(),
                    new ObjectMetadata());

            return "https://s3.amazonaws.com/casadocodigo/"
                +multipartFile.getOriginalFilename();

        } catch (AmazonClientException | IOException e) {
            throw new RuntimeException(e);
        }
```

```
    }

    private AmazonS3Client client() {
        AWSCredentials credentials = new BasicAWSCredentials(
                                        ,
                                                          );
        AmazonS3Client newClient =
                new AmazonS3Client(credentials,
                new ClientConfiguration());
        newClient.setS3ClientOptions(new S3ClientOptions()
                .withPathStyleAccess(true));
        return newClient;

    }

}
```

Perceba que temos mais códigos relativos à integração com a Amazon do que relacionado ao Spring em si. De tudo que foi escrito anteriormente, as linhas mais importantes são as que estão a seguir:

```
AmazonS3Client s3 = client();

...

s3.putObject("casadocodigo",
    multipartFile.getOriginalFilename(),
        multipartFile.getInputStream(),
        new ObjectMetadata());
```

O objeto do tipo `AmazonS3Client` encapsula toda a lógica e o protocolo de comunicação com a Amazon. O nosso único trabalho é passar os parâmetros.

- O primeiro é a pasta remota onde o arquivo vai ser salvo, também conhecido como *bucket*;
- O segundo é o nome do arquivo;

- O terceiro é alguma implementação da classe `InputStream`, que realmente representa o arquivo;
- O quarto são informações extras como: data de expiração do arquivo e qualquer outra informação que seja específica da aplicação.

A outra parte do código é a mesma para qualquer integração que você vá fazer com a Amazon. Sempre precisamos criar um objeto que contém as credenciais de acesso, para garantir que só o dono da conta possa fazer uploads para o bucket.

```
private AmazonS3Client client() {
    AWSCredentials credentials = new BasicAWSCredentials(
                                    ,
                                                       );
    AmazonS3Client newClient = new AmazonS3Client(credentials,
            new ClientConfiguration());
    newClient.setS3ClientOptions(new S3ClientOptions()
            .withPathStyleAccess(true));
    return newClient;
}
```

Basicamente essas seriam as alterações que teríamos que fazer no nosso código. Como deixamos encapsulada a lógica de gravação do arquivo, não somos obrigados a alterar nenhuma linha do nosso controller.

Simulando o S3 localmente

O empecilho com essa solução é que vamos ter que realmente criar um conta na Amazon para conseguir realizar nossos testes, o que é completamente inviável. Para resolver este problema, existe um projeto chamado *S3 Ninja*, que pode ser encontrado em http://s3ninja.net/. Eles criam este projeto com o objetivo de servir como um emulador para o S3 real, da Amazon. O S3 Ninja é uma aplicação web escrita em Java que sobe um servidor que aceita requisições vindas do próprio SDK da Amazon. Justamente o que estávamos procurando. Uma parte muito boa é que seu uso é bem simples.

O primeiro passo é realizar o download do `zip`. Basta acessar o endereço https://oss.sonatype.org/content/groups/public/com/scireum/s3ninja/2.

3/s3ninja-2.3-zip.zip. Você também pode encontrar o link no *readme* do GitHub do projeto em https://github.com/livrospringmvc/lojacasadocodigo. Uma vez que você baixou o projeto, descompacte em uma pasta da sua preferência. Ainda dentro da pasta descompactada, precisamos criar um estrutura simples de subpastas que vão simular os *buckets* da Amazon. Vamos criar primeiro a pasta `data` e, dentro dela, criaremos uma outra, chamada `s3`. A estrutura final vai ficar parecida com a que segue.
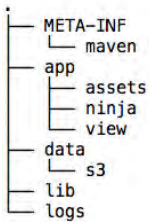
Fig. 6.3: Estrutura de pastas do s3 ninja

Agora que a estrutura está criada, basta que iniciemos o servidor do S3 Ninja.

```
java IPL
```

Perceba que `IPL` é justamente um arquivo `java` compilado que já vem na raiz do diretório descompactado. Após executar o comando, você deve ver a seguinte saída.

```
I N I T I A L   P R O G R A M   L O A D
---------------------------------------
IPL from: /Users/alberto/ambiente/servidores/s3ninja-2.3-zip
IPL completed - Loading Sirius as stage2...

Opening port 9191 as shutdown listener
```

O servidor está rodando no endereço http://localhost:9444/. Você pode acessá-lo pelo navegador.

Fig. 6.4: Tela de boas-vindas do s3 ninja

Agora que o nosso emulador está rodando, precisamos alterar o código de configuração de acesso a Amazon e fazê-lo apontar para o servidor local.

```
private AmazonS3Client client() {
    AWSCredentials credentials = new BasicAWSCredentials(
                              ,
                                                   );
    AmazonS3Client newClient = new AmazonS3Client(credentials,
            new ClientConfiguration());
    newClient.setS3ClientOptions(new S3ClientOptions()
            .withPathStyleAccess(true));

    //nova linha
    newClient.setEndpoint("http://localhost:9444/s3");
    return newClient;

}
```

Pronto! Quando cadastrarmos um novo livro, o arquivo do sumário vai ser gravado na pasta `data/s3/casadocodigo/nomedoarquivo`, no local que você escolheu para executar o S3 Ninja. Perceba que também alteramos o retorno do método `write`, da classe `FileSaver`.

```
public String write(String baseFolder,
MultipartFile multipartFile) {
    AmazonS3Client s3 = client();
```

```
    try {
        s3.putObject("casadocodigo",
            multipartFile.getOriginalFilename(),
                multipartFile.getInputStream(),
                new ObjectMetadata());

        //url de acesso ao arquivo
        return "https://s3.amazonaws.com/casadocodigo/"
         +multipartFile.getOriginalFilename()+"?noAuth=true";

    } catch (AmazonClientException | IOException e) {
        throw new RuntimeException(e);
    }

}
```

A ideia é que é nosso método já retorne o endereço de acesso ao arquivo enviado, para que possamos usá-lo nos links do nosso sistema.

Isolando a criação do AmazonS3Client

A nossa alteração nos levou a deixar o código de criação do objeto de acesso à Amazon misturado com o código da classe `FileSaver`. Como já discutimos quando recebemos injetado o DAO, o nosso objetivo nesse código é usar o cliente da Amazon e não ficar criando. Portanto, vamos começar recebendo um objeto do tipo `AmazonS3Client`, em vez de criá-lo.

```
@Component
    public class FileSaver {

        @Autowired
        private AmazonS3Client s3;

        public String write(String baseFolder,
        MultipartFile multipartFile) {
            try {
                s3.putObject("casadocodigo",
                    multipartFile.getOriginalFilename(),
                        multipartFile.getInputStream(),
```

```
                    new ObjectMetadata());

                return "http://localhost:9444/s3/casadocodigo/"
                +multipartFile
                .getOriginalFilename()+"?noAuth=true";

            } catch (AmazonClientException | IOException e) {
                throw new RuntimeException(e);
            }

        }
    }
```

Para este código funcionar, precisamos criar uma classe de configuração que construa o objeto para nós.

```
package br.com.casadocodigo.loja.conf;

import org.springframework.context.annotation.Bean;

import com.amazonaws.ClientConfiguration;
import com.amazonaws.auth.AWSCredentials;
import com.amazonaws.auth.BasicAWSCredentials;
import com.amazonaws.services.s3.AmazonS3Client;
import com.amazonaws.services.s3.S3ClientOptions;

public class AmazonConfiguration {

    @Bean
    public AmazonS3Client s3Ninja() {
        AWSCredentials credentials = new BasicAWSCredentials(
                                        ,
                                                            );
        AmazonS3Client newClient =
            new AmazonS3Client(credentials,
            new ClientConfiguration());
        newClient.setS3ClientOptions(new S3ClientOptions()
                .withPathStyleAccess(true));
        newClient.setEndpoint("http://localhost:9444/s3");
```

```
        return newClient;

    }
}
```

Perceba que é uma classe normal, apenas criamos um método e usamos anotação `@Bean` para dizer que ele produz um objeto que deve ser gerenciado pelo Spring. O último passo é ensinar ao Spring que essa classe existe e, para isso, vamos alterar a classe `ServletSpringMVC`.

```
package br.com.casadocodigo.loja.conf;

...

public class ServletSpringMVC extends
AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class[] { SecurityConfiguration.class,
            AppWebConfiguration.class,
            JPAConfiguration.class,
            AmazonConfiguration.class };
    }

    ...
```

Agora você tem duas opções para salvar os arquivos enviados pelos seus usuários. Este autor sugere que você sempre guarde o arquivo em um local fora do servidor web. A complexidade adicionada ao projeto é compensada pela flexibilidade de manter os arquivos entre reloads da aplicação, em tempo de desenvolvimento. Além disso, a instalação no ambiente de produção vai ser facilitada, já que você não vai ter que se preocupar em manter os arquivos que já tenham sido enviados pelos usuários. Tudo vai estar em um lugar separado.

6.5 Conclusão

Neste capítulo, focamos unicamente no upload de arquivos. Essa é uma funcionalidade muito comum nas aplicações, mas muitas vezes esquecemos de olhar com detalhes, para entender exatamente como funciona. Espero que este capítulo tenha ajudado a deixar as configurações um pouco mais claras para você.

Não pare agora, já vá para o próximo capítulo onde vamos simular o fechamento de uma compra e a integração com um outro sistema. Também vamos ver como podemos ajudar o servidor a processar requests mais longos.

Capítulo 7

Carrinho de compras

Já temos o cadastro de livros e a listagem dos mesmos na loja da Casa do Código, chegou o momento de implementarmos o processo de compra. Para começarmos essa fase, precisamos, primeiramente, possibilitar que os usuários escolham os livros que mais lhes interessam e os coloquem em um carrinho de compras.

7.1 URI templates

Vamos tentar seguir exatamente o mesmo processo usado no site da Casa do Código. Por lá, para o visitante do site colocar um livro no carrinho, ele precisa navegar até a página de detalhes do livro.

Fig. 7.1: Página de detalhes do livro

Para seguir a mesma ideia, vamos criar a nossa página de detalhes do livro. Aqui, para ficar ainda mais claro que estamos literalmente reproduzindo a aplicação, será usado o mesmo HTML da Casa do Código. Como o código da página é relativamente grande, o leitor pode navegar até o link https://github.com/livrospringmvc/lojacasadocodigo/blob/master/src/main/webapp/WEB-INF/views/products/show.jsp para ver a página completa. No trecho a seguir, veja algumas partes importantes.

```
<header id="product-highlight" class="clearfix">
    <div id="product-overview" class="container">
        <img itemprop="image" width="280px" height="395px"
            src=’’
            class="product-featured-image"
            alt="${product.title}">
        <h1 class="product-title" itemprop="name">
            ${product.title}
        </h1>
        <p class="product-author">
            <span class="product-author-link">
                ${product.title}
```

```
            </span>
        </p>

        <p itemprop="description" class="book-description">
        ${product.description}
         Veja o <a
                href="<c:url value='/${product.summaryPath}'/>"
                target="_blank">sum&#225;rio</a> completo do
                livro!
        </p>
    </div>
</header>

<section class="buy-options clearfix">
    <form action="<c:url value="/shopping"/>" method="post"
        class="container">
        <input type="hidden" value="${product.id}"
            name="productId"/>
        <ul id="variants" class="clearfix">
            <c:forEach items="${product.prices}" var="price">
                <li class="buy-option">

                    <input type="radio"
                    name="bookType" class="variant-radio"
                        id="${product.id}-${price.bookType}"
                    value="${price.bookType}"
                    ${price.bookType.name()=='COMBO'?'checked':''
                    }>

                    <label  class="variant-label"
                        for="${product.id}-${price.bookType}">
                        ${price.bookType}
                    </label>
                    <p class="variant-price">${price.value}</p>
                </li>
            </c:forEach>
        </ul>

        <input type="submit" class="submit-image icon-basket-alt"
```

```
            alt="Compre agora"
            title="Compre agora '${product.title}'!"
            value="comprar"/>
    </form>
</section>
```

Apenas para você não ficar perdido, o nome desse arquivo é `show.jsp` e fica na pasta `WEB-INF/views/products`.

Perceba que simplesmente usamos as informações que já possuíamos sobre um determinado produto. O único ponto para o qual ainda não fizemos o suporte é para a URL `/shopping`, cujo controller vai ser responsável por adicionar o produto no carrinho de compras. Ainda não se preocupe com ela, vamos resolver isso no momento correto. Um outro detalhe com que não nos preocupamos foi o de recuperar os arquivos de estilos usados pela Casa do Código, estamos apenas reaproveitando a estrutura do HTML.

Agora que já temos a página, é necessário construir o método do controller que carrega o produto. Faremos isso na classe `ProductsController`.

```
@RequestMapping(method=RequestMethod.GET,value="/show")
public ModelAndView show(Integer id){
    ModelAndView modelAndView =
        new ModelAndView("products/show");
    Product product = productDAO.find(id);
    modelAndView.addObject("product", product);
    return modelAndView;
}
```

Precisamos também adicionar um link na listagem de produtos, para que os usuários possam fazer a navegação pela nossa aplicação.

```
...

<c:forEach items="${products}" var="product">
    <tr>
        <td>
            <a
            href="${spring:mvcUrl('PC#show').arg(0,product.id)
            .build()}">
```

```
                ${product.title}
            </a>
        </td>
        <td>
            <c:forEach items="${product.prices}" var="price">
                [${price.value} - ${price.bookType}]
            </c:forEach>
        </td>
    </tr>
</c:forEach>

...
```

Mais uma vez fizemos uso da função `mvcUrl`, para que ela nos ajude a montar a URL. A única diferença é que, antes de invocar o método `build()`, tivemos que chamar o `arg(indice,valor)` para que o endereço possa ser montado utilizando corretamente os parâmetros.

O endereço que aparece, após o usuário clicar no link que leva para a página de detalhe do produto, é o `casadocodigo/produtos/show?id=2`. Na verdade, essa é a maneira padrão de passar parâmetros via get só que, se olharmos para o mesmo endereço no site da Casa do Código, veremos que ele possui uma estrutura um pouco diferente, http://www.casadocodigo.com.br/products/livro-apis-java.

O trecho `livro-apis-java`, funciona basicamente como um id dentro do sistema da Casa do Código. Ao contrário da nossa implementação, esse parâmetro vai como parte da própria URL, técnica conhecida como **URI Template**.

E qual o motivo de passarmos o parâmetro de uma maneira diferente? Os motores de busca, tipo o Google, preferem endereços que, se acessados, sempre retornem o mesmo resultado. E esta é a diferença: quando usamos a `?` estamos dizendo que basta uma mudança no valor do parâmetro que o resultado da página muda e, quando usamos o parâmetro como parte da URI, isso passa a ideia de que aquele é um endereço fixo.

É muito simples aplicarmos essa mudança no nosso projeto.

```
@RequestMapping("/{id}")
public ModelAndView show(@PathVariable("id") Integer id){
```

```
    ModelAndView modelAndView =
        new ModelAndView("products/show");
    Product product = products.find(id);
    modelAndView.addObject("product", product);
    return modelAndView;
}
```

Usamos a notação `{nomeDoParametro}` para montar a URL com argumentos misturados. Além disso, temos que usar a annotation `PathVariable` para indicar o parâmetro do nosso método que deve ser populado com a respectiva parte da URL. Caso tivéssemos agrupado os livros por categorias, poderíamos ter algo parecido com o que segue.

```
//apenas um exemplo

@RequestMapping("/{categoryId}/{productId}")
public ModelAndView show(@PathVariable("categoryId")
  Integer categoryId, @PathVariable("productId") Integer id){

    ModelAndView modelAndView =
        new ModelAndView("products/show");
    Product product = products.find(id);
    modelAndView.addObject("product", product);
    return modelAndView;
}
```

Um detalhe importante é que os parâmetros de URL só podem ser associados com tipos considerados mais simples, por exemplo:

- tipos primitivos
- String
- Date
- Calendar

7.2 Carrinho de compras e o escopo de Sessão

Agora, que já conseguimos navegar para a página de detalhe do livro, podemos começar a implementar a funcionalidade de adicioná-los ao carrinho de compras. Vamos apenas relembrar o trecho da JSP responsável por isso.

```
<section class="buy-options clearfix">
    <form action="<c:url value="/shopping"/>" method="post"
        class="container">
        <input type="hidden" value="${product.id}"
         name="productId"/>
        <ul id="variants" class="clearfix">
            <c:forEach items="${product.prices}" var="price">
                <li class="buy-option">

                    <input type="radio"
                    name="bookType" class="variant-radio"
                        id="${product.id}-${price.bookType}"
                    value="${price.bookType}"
                    ${price.bookType.name()=='COMBO'?'checked':''
                    }>

                    <label  class="variant-label"
                        for="${product.id}-${price.bookType}">
                        ${price.bookType}
                    </label>
                    <p class="variant-price">${price.value}</p>
                </li>
            </c:forEach>
        </ul>

        <input type="submit" class="submit-image icon-basket-alt"
            alt="Compre agora"
            title="Compre agora '${product.title}'!"
            value="comprar"/>
    </form>
</section>
```

A URL que deve ser acessada é a `/shopping` e, além do ID do produto, precisamos saber qual o tipo de livro que o usuário pretende

comprar. Para isolar essa lógica, vamos criar um novo controller, o `ShoppingCartController`.

```
@Controller
@RequestMapping("/shopping")
public class ShoppingCartController {

    @Autowired
    private ProductDAO productDAO;
    @Autowired
    private ShoppingCart shoppingCart;

    @RequestMapping(method=RequestMethod.POST
    public ModelAndView add(Integer productId,BookType bookType){
        ShoppingItem item = createItem(productId, bookType);
        shoppingCart.add(item);
        return new ModelAndView("redirect:/produtos");
    }

    private ShoppingItem createItem(Integer productId,
      BookType bookType) {
        Product product = productDAO.find(productId);
        ShoppingItem item = new ShoppingItem(product,bookType);
        return item;
    }
```

Aqui não tem nenhum código que você já não tenha visto no livro: um controller normal, recebendo suas dependências injetadas e os métodos que tratam as requisições dos usuários. As classes novas são as que representam o carrinho de compras, no caso a `ShoppingCart` e a `ShoppingItem`, que representa o livro sendo adicionado no carrinho. Não vamos perder tempo dissecando a lógica dessas classes, vamos ficar atentos apenas às partes que estejam ligadas ao Spring. Caso queira conferir o código completo delas, basta acessar o endereço http://tinyurl.com/pajz3dh.

É necessário avisar o Spring de que queremos injetar um objeto do tipo `ShoppingCart` em outros objetos do nosso código.

```
@Component
public class ShoppingCart {
```

```
    ....
}
```

Basta adicionar a annotation `@Component` e, além disso, precisamos alterar a classe `AppWebConfiguration` e pedir que o pacote dessa classe passe a ser escaneado.

```
@EnableWebMvc
@ComponentScan(basePackageClasses={HomeController.class,
    ProductDAO.class,FileSaver.class,ShoppingCart.class})
public class AppWebConfiguration {
    ...
}
```

Adicionando produtos no carrinho

Já podemos começar a adicionar os itens dentro do nosso carrinho e, por sinal, tudo parece estar correndo bem. O valor que está sendo impresso dentro do nosso controller cresce a cada novo livro que é adicionado. Agora será feito um teste, vamos abrir uma nova página do seu navegador usando o famoso *Private Browsing*. Dessa maneira o navegador não salva nem reaproveita nenhuma informação que já existia previamente.

- No Chrome, pressione `Ctrl + Shift + N`
- No Firefox, pressione `Ctrl + Shift + P`

Agora tente adicionar mais um livro ao carrinho e confira o valor impresso. Provavelmente, apareceu o valor incrementado, como se só existisse um carrinho para todos usuários do site! Isso acontece porque, por default, o Spring mantém os objetos gerenciados por ele no escopo conhecido como `Application`. Ou seja, ele cria apenas uma instância e a reaproveita durante toda a execução do programa. Para ficar mais claro, você poderia fazer assim:

```
@Component
@Scope(value = WebApplicationContext.SCOPE_APPLICATION)
public class ShoppingCart {
    ...
}
```

O que necessitamos é de que exista uma instância do carrinho para cada **sessão** de navegação que existir naquele momento. Esse é o escopo mais conhecido como **Session**.

```
@Component
@Scope(value = WebApplicationContext.SCOPE_SESSION)
public class ShoppingCart {
    ...
}
```

Mudar o escopo é muito simples, basta passarmos o valor que queremos através da annotation `@Scope`. Outro escopo muito comum em aplicações web é o de *Request*, que você pode usar com a constante `WebApplicationContext.SCOPE_REQUEST`.

Injetando objetos de escopos menores

Caso você tenha tentado subir a aplicação, deve ter percebido que foi lançada uma exception com uma mensagem um tanto quanto grande. Vamos ver apenas uma parte dela.

```
No thread-bound request found: Are you referring to request
attributes outside of an actual web request ...
```

A exception está nos informando de que estamos tentando usar um objeto de escopo relativo à web em outro objeto cujo escopo não tem nada a ver com a web. E é justamente o que está acontecendo, a classe `ShoppingCartController` não tem nenhuma anotação relativa ao escopo e, como já vimos, o default é o de `Application`. Nesta situação, o Spring não consegue resolver a injeção, pois um objeto de escopo de aplicação não tem nada a ver com os escopos web em si. A regra geral é: injete sempre objetos de escopos maiores em objetos de escopos menores, nunca ao contrário.

- Objetos no escopo `Application` podem ser injetados em qualquer lugar.

- Objetos no escopo de `Session` podem ser injetados em outros do mesmo escopo ou em escopo de `Request`.

Pensando nisso, a maneira mais fácil de resolver esse problema é modificando a classe `ShoppingCartController` e a transformando em objeto de escopo de `Request`.

```
@Controller
@RequestMapping("/shopping")
@Scope(value=WebApplicationContext.SCOPE_REQUEST)
public class ShoppingCartController {
    ...
}
```

A opinião deste autor é de que o Spring MVC falhou nesse quesito. Pensando em uma aplicação web, faz todo o sentido que os objetos que representam controllers sejam de escopo de requisição. Afinal de contas, essa é a natureza de uma aplicação web; objetos são criados e destruídos ao fim de uma requisição. Só que querer não é poder, então temos que nos adequar ao que o framework nos fornece.

Como o escopo de Aplicação é muito comum em aplicações que usam o Spring, eles forneceram outra maneira de resolver o problema, bem mais obscura.

```
@Component
@Scope(value = WebApplicationContext.SCOPE_SESSION,
    proxyMode = ScopedProxyMode.TARGET_CLASS)
public class ShoppingCart {
 ...
}
```

Podemos usar o atributo `proxyMode` para indicar como os objetos gerenciados devem ser criados. O modo default é criar uma instância normal, via reflection mesmo. Algo parecido com o que segue.

```
Class klass = //recupera classe do objeto
klass.newInstance();
```

E esse é justamente o motivo do problema, já que ele só vai poder criar o objeto do tipo `ShoppingCart` quando existir uma requisição web. Quando definimos o *proxyMode* igual a `ScopedProxyMode.TARGET_CLASS`, estamos pedindo que o Spring modifique a maneira como ele cria o objeto, neste

caso usando uma biblioteca especializada nisso, chamada `cglib`. Usando esse artifício, conseguimos injetar um objeto de escopo menor, no caso de sessão, em um maior, no caso de aplicação.

Proxy, um Design Pattern

O nome do atributo é `proxyMode` não por acaso. Ele avisa ao Spring que o Design Pattern Proxy deve ser aplicado para a criação dos objetos. Basicamente, a implementação desse pattern faz com que, em vez de trabalharmos com o objeto alvo, neste caso o `ShoppingCart`, seja criada uma segunda classe que controla todo o acesso ao original. O próprio dá suporte à criação de um Proxy, só que ele impõe que sejam criadas interfaces para isso. Através da `cglib`, conseguimos a mesma implementação, só que baseada na classe concreta. O Spring, como era esperado, suporta as duas maneiras.

Entendo que usar o proxy é o jeito mais comum do mercado de atingir a injeção, mas neste caso continuo achando que colocar o escopo de `request` no controller é a solução mais limpa. Fique à vontade e decida qual solução se encaixa melhor no seu problema.

Pronto, agora nossos usuários podem fazer suas compras sem que um coloque itens no carrinho do outro.

7.3 Exibindo os itens do carrinho

Na página de detalhe do livro, além de podermos colocá-lo no carrinho, também é possível exibir o número de itens que já existem atualmente.

```
<ul class="clearfix">

    <li><a href=""
        rel="nofollow">Seu carrinho (${shoppingCart.quantity})
    </a></li>

    ....
```

```
</ul>
```

Lembre-se que esse é mais um trecho da mesma página de detalhe, seu código completo você pode ver direto no Github, basta seguir este link: http://tinyurl.com/odjn4j3

O problema aqui é que, sempre que a página de detalhe é acessada, o trecho que deveria exibir a quantidade de itens não exibe nada.

```
Seu carrinho ()
```

A complicação aqui é que o Spring MVC está disponibilizando o objeto `ShoppingCart` com outro nome na Expression Language.

```
${sessionScope['scopedTarget.shoppingCart'].quantity}
```

A chave criada foi a `scopedTarget.shoppingCart` e, como tem um `.` (ponto) no nome, somos obrigados a acessar a chave através da sintaxe de mapa. Apenas por curiosidade, a variável com o nome `sessionScope` está sempre disponível nas JSPs para que que você tenha acesso diretamente ao mapa de variáveis que estejam neste escopo. O mesmo acontece com os objetos que estão no request, que ficam disponíveis através do `$requestScope`. Só que a pergunta que não quer calar é: por que o Spring MVC colocou a variável começando com `scopedTarget.`?

O problema foi justamente a nossa abordagem para resolver a injeção do carrinho de compras. Como foi escolhida a estratégia baseada no proxy, o framework coloca os objetos no escopo com esse prefixo, justamente para poder dar um tratamento diferenciado internamente. Caso tivéssemos optado pela abordagem de trocar o escopo do nosso controller pelo request e, não houvesse a necessidade de trocar a maneira como criamos os objetos do tipo `ShoppingCart`, o Spring MVC teria colocado na sessão a variável com o mesmo nome da classe e nossa JSP funcionaria tranquilamente. Lembre-se sempre que tudo é uma troca: quando decidimos ir por um caminho temos que ter em mente que, talvez, um preço seja cobrado.

De toda forma, agora nosso problema está resolvido. Caso não tenha gostado dessa abordagem, o que é completamente compreensível, podemos tentar outros caminhos.

Expondo os beans na Expression Language

Podemos ensinar ao Spring MVC que queremos que todos os nossos objetos gerenciados fiquem disponíveis para uso, através da Expression Language.

```
...
public class AppWebConfiguration {

    @Bean
    public InternalResourceViewResolver
    internalResourceViewResolver() {
      InternalResourceViewResolver resolver =
          new InternalResourceViewResolver();
      resolver.setPrefix("/WEB-INF/views/");
      resolver.setSuffix(".jsp");
      //essa linha expõe todo mundo
      resolver.setExposeContextBeansAsAttributes(true);
      return resolver;
    }
    ...
}
```

Com essa alteração o trecho `${shoppingCart}` já vai funcionar. Ele vai usar o nome que está registrado dentro do *container* do Spring, que por default é o mesmo da classe. Essa solução até funciona, a única preocupação é se alguém começar a acessar tudo que é objeto a partir da JSP.

Para restringir um pouco a fanfarra, pode-se liberar apenas alguns objetos para serem expostos.

```
@Bean
public InternalResourceViewResolver
 internalResourceViewResolver() {
  InternalResourceViewResolver resolver =
      new InternalResourceViewResolver();
  resolver.setPrefix("/WEB-INF/views/");
```

```
  resolver.setSuffix(".jsp");
  //passamos o exato nome da classe que será registrada
  resolver.setExposedContextBeanNames("shoppingCart");
  return resolver;
}
```

Dessa maneira, você pode controlar quais objetos devem estar sempre disponíveis.

7.4 Conclusão

Agora já conseguimos manter objetos em escopos diferentes e acessá-los nas nossas páginas. Neste momento do livro, você já tem ferramentas suficientes para começar a fazer sua mais nova aplicação! Não perca tempo, já comece a pensar no próximo projeto para praticar tudo que foi estudado até agora. Caso não esteja cansado, já leia o próximo capítulo, pois vamos fazer uma integração com um sistema externo, justamente para fechar o processo de compra do nosso usuário.

Capítulo 8

Retornos assíncronos

Nosso usuário já está quase fechando a compra dele. Agora, com os livros já colocados no carrinho, só falta que ele preencha os dados de pagamento e efetivamente compre os livros. Aqui, vamos apenas simplificar o processo e deixar que nosso usuário compre um livro apenas apertando um botão.

```
<tbody>
    <c:forEach items="${shoppingCart.list}" var="item">
    <tr>
        <td class="cart-img-col">
            <img src="" alt="${item.product.title}"/>
        </td>
        <td class="item-title">
            ${item.product.title} - ${item.bookType}
        </td>
        <td class="numeric-cell">
```

```
            ${item.price}
        </td>
        <td class="quantity-input-cell">
            <input type="number" min="0" readonly="readonly"
            value="${shoppingCart.getQuantity(item)}">
        </td>
        <td class="numeric-cell">
            ${shoppingCart.getTotal(item)}
        </td>
    </tr>
    </c:forEach>

    </tbody>
    <tfoot>
    <tr>
        <td colspan="2">
            <form
                action="${spring:mvcUrl('PC#checkout').build()}"
                method="post">
                    <input type="submit" class="checkout"
                    name="checkout" value="Finalizar compra "
                    id="checkout"/>
            </form>
        </td>
        <td class="numeric-cell">
            ${shoppingCart.total}
        </td>
        <td></td>
    </tr>
    </tfoot>
```

Essa é a página onde nossos usuários poderão ver os itens adicionados ao carrinho. O nome dela é `items.jsp` e vai ficar na pasta `WEB-INF/views/shoppingCart`. Lembre-se que, para conferir o código completo, basta que você navegue até o repositório do projeto, https://github.com/livrospringmvc/lojacasadocodigo. Como é de praxe, nosso usuário não pode navegar diretamente para essa página, o endereço de acesso deve ser referente a um método na nossa classe `ShoppingCartController`.

```
@RequestMapping(method=RequestMethod.GET)
public String items(){
    return "shoppingCart/items";
}
```

Todo o código presente nesse JSP é bem normal, não tem nada que não foi estudado até agora. O único detalhe novo é a presença do formulário de checkout, que aponta para o método `checkout` da classe `PaymentController`, exatamente o código que vamos começar a construir.

```
@Controller
@RequestMapping("/payment")
public class PaymentController {

    @Autowired
    private ShoppingCart shoppingCart;

    @RequestMapping(value="checkout",method=RequestMethod.POST)
    public String checkout() {
            BigDecimal total = shoppingCart.getTotal();
            //código de integração
            return "redirect:/success";
    }

}
```

É um controller comum, o único detalhe que vamos ter que implementar agora é a integração com o meio de pagamento. Para deixar a aplicação mais real, foi criada uma aplicação web chamada **book-payment**, que está hospedada atualmente no Heroku.

Caso queira efetuar um teste e estiver no Linux ou Mac, basta executar um *curl*.

```
curl --header "Content-type: application/json" --request POST
    --data '{"value": 600}'
    http://book-payment.herokuapp.com/payment
```

A lógica é simples: só aceitamos pagamentos de até 500 reais. Somos obrigados a passar os dados usando o `JSON`, um formato de dados bastante

usado em aplicações que realizam integrações via HTTP. Caso o valor tenha seguido a regra, é retornado o status *201*, que indica que um novo recurso foi criado, neste caso, o pagamento. Por outro lado, se tiver sido passado um valor maior que 500, é retornado o status *400*, o que indica que a requisição possui dados inválidos.

RestTemplate

Vamos precisar de uma biblioteca que nos ajude a realizar a requisição HTTP, para que nossa integração funcione. A lib mais famosa é a **Http Components**, que pode ser encontrada aqui http://hc.apache.org.

Para não termos que lidar com os detalhes da biblioteca, vamos utilizar um objeto que já encapsula toda essa lógica.

```
@Controller
@RequestMapping("/payment")
public class PaymentController {

    @Autowired
    private ShoppingCart shoppingCart;
    @Autowired
    private RestTemplate restTemplate;

    @RequestMapping(value = "checkout",
    method = RequestMethod.POST)
    public String checkout() {
        BigDecimal total = shoppingCart.getTotal();

        String uriToPay =
            "http://book-payment.herokuapp.com/payment";
        try {
            String response =
                restTemplate.postForObject(uriToPay,
                new PaymentData(total), String.class);
            return "redirect:/payment/success";
        } catch (HttpClientErrorException exception) {
            return "redirect:/payment/error";
        }
```

```
    }

}
```

O objeto do tipo `RestTemplate` já disponibiliza diversos métodos, que podemos usar para realizar diversos tipos de requisições. Perceba que foi muito simples, apenas apontamos o endereço e invocamos o método que realiza o `POST` e espera como resposta uma simples String. Para conseguirmos a injeção deste objeto, precisamos também alterar a nossa classe `AppWebConfiguration` para registrar o bean no *container* do Spring.

```
@Bean
public RestTemplate restTemplate(){
    return new RestTemplate();
}
```

Outro detalhe que pode chamar a atenção é a utilização da classe `PaymentData`. Qual o motivo de necessitarmos dela? O sistema de pagamento com que estamos realizando a integração pede que o formato do dado passado seja o seguinte:

```
{"value": 600}
```

É como se fosse um *Map*, precisamos de uma chave chamada `value` associada a um valor qualquer. Caso a gente passe o `BigDecimal` direto, qual seria o nome dessa chave? É justamente por isso que precisamos criar uma classe.

```
package br.com.casadocodigo.loja.models;

import java.math.BigDecimal;

public class PaymentData {

    private BigDecimal value;

    public PaymentData() {
    }
```

```
    public PaymentData(BigDecimal value) {
        this.value = value;
    }

    public BigDecimal getValue() {
        return value;
    }

}
```

Repare que temos um atributo que se chama exatamente **value**, justamente para que o Spring MVC possa pegar o objeto e gerar o `JSON` com as chaves corretas.

Configurando o converter para JSON

Neste momento, o nosso código de integração, se testado, lançará uma exception.

```
org.springframework.web.client.RestClientException:
    Could not write request: no suitable HttpMessageConverter
    found for request type
    [br.com.casadocodigo.loja.models.PaymentData]
```

O Spring MVC está nos dizendo que não conseguiu achar uma maneira de converter o objeto do tipo `PaymentData` para o `JSON`. Como já estamos vendo faz um tempo, ele é muito modular e, para tudo de que necessitamos, temos que ir adicionando as dependências necessárias. Nesse caso, precisamos adicionar uma biblioteca chamada Jackson, muita famosa no mundo Java no que diz a respeito à transformação de objetos em `JSON`. Precisamos realizar a alteração a seguir, no nosso arquivo `pom.xml`.

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-core</artifactId>
    <version>2.5.1</version>
</dependency>
```

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.5.1</version>
</dependency>
```

Pronto, agora já conseguimos realizar a nossa integração sem maiores problemas!

8.1 Executando operações demoradas assincronamente

Agora que nosso código de integração já funciona, vamos tentar pensar um pouco adiante. A Casa do Código é uma empresa de sucesso, o Brasil inteiro compra livros de tecnologia nela. Em certos momentos do ano, como na Black Friday, ela libera certas promoções que fazem com que o site fique muito movimentado e, com isso, temos muitos usuários concluindo compras ao mesmo tempo. Algo similar acontece em todas outras empresas.

Não é raro, nessas situações de uso intenso, que os sistemas fiquem um pouco mais lentos ou até que caiam e fiquem fora do ar por um tempo, o que pode gerar um prejuízo grande. Em geral todos os sistemas têm seus pontos de gargalo e, como não poderia ser diferente, nós temos o nosso também. Já se perguntou quanto tempo pode levar para a nossa aplicação fazer uma requisição para o sistema de pagamento? Caso esse outro sistema demore, o que pode acontecer com o nosso site?

Neste exato momento o funcionamento é basicamente o seguinte. Quando uma nova requisição chega no nosso Tomcat, ele faz um tratamento inicial e delega a responsabilidade para a Servlet configurada, nesse caso a do Spring MVC. A `DispatcherServlet` vai descobrir qual dos nossos controllers deve tratar a requisição e fazer o trabalho para que isso realmente aconteça. Todo esse fluxo, acontece dentro de uma Thread criada pelo Tomcat, para que ele possa tratar várias requisições simultaneamente. Essa Thread só é liberada quando o método do nosso controller acaba de fazer o trabalho e indica para qual endereço devemos ir. Esse modelo de trabalho, onde alguém fica esperando o outro acabar, para aí sim continuar seu trabalho, é conhe-

cido como modelo **síncrono**. O problema desse modelo é que o Tomcat tem um número limitado de Threads que podem ser criadas e, caso esse número chegue no limite, as requisições dos nossos usuários vão começar a ser enfileiradas e podemos ter uma lentidão nas aplicações.

Como já comentamos, em um pico de acesso, podemos sofrer deste problema justamente no momento em que precisamos realizar a integração com um sistema externo, no qual não temos nenhum controle sobre o tempo de resposta.

Indo para o assíncrono

A grande sacada para esse tipo de situação é tentar liberar o Tomcat para atender outras requisições enquanto realizamos a integração com o nosso meio de pagamento. É exatamente para isso que, desde a versão 3 da especificação, você pode criar uma Servlet que consegue trabalhar de modo assíncrono! E, para a nossa sorte, a Servlet do Spring MVC já vem preparada para trabalhar desse jeito.

```
@RequestMapping(value = "checkout", method = RequestMethod.POST)
public Callable<String> checkout() {
    return () -> {
        BigDecimal total = shoppingCart.getTotal();
        String uriToPay = "http://localhost:9000/payment";
        try {
            String response =
                restTemplate.postForObject(uriToPay,
                    new PaymentData(total), String.class);
            return "redirect:/payment/success";
        } catch (HttpClientErrorException exception) {
            return "redirect:/payment/error";
        }

    };
}
```

Basta que retornemos um objeto do tipo `Callable`, que existe desde o Java 5 e é um análogo ao objeto tipo `Runnable`, muito comum para quem usa Threads. A única diferença do `Callable` é que ele nos permite dar um

retorno, algo necessário para nós, já que precisamos informar para qual endereço vamos depois da integração. Só de você retornar um `Callable`, o Spring MVC já vai criar iniciar um contexto **assíncrono** em sua Servlet e liberar para que o Tomcat possa usar a Thread dele para atender novas requisições. Assim, quando recebermos a resposta da integração, retornamos o endereço de redirect e o mesmo é informado para o Tomcat. Perceba que tudo ocorre de maneira bem transparente, não precisamos em nenhum momento lidar com a Servlet assíncrona em si, só precisamos estar cientes de que isso só é possível porque a especificação teve a preocupação.

É importante notar que não estamos tratando da performance do nosso site, provavelmente o tempo de requisição vai ser o mesmo. O que estamos tratando aqui é da escalabilidade, estamos tentando manter o tempo médio de requisição mesmo que a aplicação sofra com um acesso acima do normal. Você pode usar essa técnica para todo método do controller que tiver de realizar uma integração com um sistema que você não conhece. Um outro exemplo seria o envio de e-mail.

Sintaxe do Java 8 para classes anônimas

Usamos a sintaxe dos **lambdas**, disponível no Java 8, para retornarmos um objeto do tipo `Callable`. Caso você esteja utilizando uma versão inferior do Java, pode trocar aquele código pelo seguinte:

```
return new Callable<String>() {
    @Override
    public String call() throws Exception {
        BigDecimal total = shoppingCart.getTotal();
        String uriToPay = "http://localhost:9000/payment";
        try {
            String response =
                restTemplate.postForObject(uriToPay,
                new PaymentData(total), String.class);
            return "redirect:/payment/success";
        } catch (HttpClientErrorException exception) {
            return "redirect:/payment/error";
        }
    }
};
```

Que era o comum, quando se fazia necessário o uso de classes anônimas.

8.2 DefferedResult e um maior controle sobre a execução assíncrona

O uso do `Callable` é a maneira mais simples de realizar a execução de um código assíncrono dentro do nosso controller. Talvez o único ponto negativo é que você não tem nenhum controle de como esse código vai ser executado. Por exemplo:

- Quantas novas Threads existem disponíveis para a execução do código assíncrono?
- E se eu quiser controlar a criação das Threads?

- E se eu precisar rodar esse código dentro de uma fila de processamento?

Caso você se veja em algumas dessas situações, podemos usar o objeto do tipo `DefferedResult` que, em uma tradução livre, seria algo parecido como retorno postergado.

```
@RequestMapping(value = "checkout", method = RequestMethod.POST)
public DeferredResult<String> checkout() {

    BigDecimal total = shoppingCart.getTotal();
    DeferredResult<String> result = new DeferredResult<String>();

    IntegrandoComPagamento integrandoComPagamento =
        new IntegrandoComPagamento(result, total, restTemplate);

    Thread thread = new Thread(integrandoComPagamento);
    thread.start();
    return result;
}
```

Respire, pois muita coisa aconteceu nesse código. Estou aqui justamente para explicar cada um dos passos. A parte principal é que criamos um objeto do tipo `DefferedResult` e o passamos como argumento para a construção de um outro objeto, que vai executar em uma Thread diferente e, em algum momento, vai sinalizar que acabou de realizar o processamento. Essa realmente é a parte mais importante desse código. Sempre que você for trabalhar com o `DeferredResult`, o objeto que foi retornado é o mesmo que vai ser passado como argumento para a execução em outra Thread.

No código anterior, extraímos a integração com o meio de pagamento para uma outra classe, cujo código vai ser executado na Thread que criamos manualmente:

```
package br.com.casadocodigo.loja.service;

public class IntegrandoComPagamento implements Runnable {

    private DeferredResult<String> result;
    private BigDecimal value;
```

```
    private RestTemplate restTemplate;

    public IntegrandoComPagamento(DeferredResult<String> result,
            BigDecimal value, RestTemplate restTemplate) {
        super();
        this.result = result;
        this.value = value;
        this.restTemplate = restTemplate;
    }

    @Override
    public void run() {
        String uriToPay = "http://localhost:9000/payment";
        try {
            String response =
                restTemplate.postForObject(uriToPay,
                new PaymentData(value), String.class);
            //linha de notificação
            result.setResult("redirect:/payment/success");
        } catch (HttpClientErrorException exception) {
            result.setResult("redirect:/payment/error");
        }
    }

}
```

O método `setResult` é parte importante desse trecho de código. Quando você o invoca, o objeto do tipo `DeferredResult` notifica o SpringMVC de que o contexto assíncrono acabou e que ele pode gerar o retorno para o usuário da aplicação.

O uso do `DeferredResult` ainda permite um controle mais fino da execução assíncrona, disponibilizando alguns outros métodos que podem ser úteis.

- `onTimeout(Runnable callback)`: registre um objeto que deve ter o método invocado caso expire um determinado tempo.
- `onCompletion(Runnable callback)`: registre um objeto que deve ter o método invocado quando a execução acabar. Por exemplo,

registrar um log.

8.3 Conclusão

Neste capítulo conseguimos realizar uma integração com um sistema de terceiro e, para ficar ainda melhor, a integração pode ser feita sem travar o nosso servidor. Você deve levar essa lição para a sua vida, quando for realizar qualquer tipo de operação que envolva um outro sistema que você não conhece; nunca fique faça nenhuma comunicação de maneira blocante. É uma pena que as APIs disponíveis no mundo Java já não sejam todas pensadas dessa maneira. Imagine se a especificação JDBC obrigasse todos os drivers a realizarem as consultas de maneira assíncrona, tudo seria muito melhor.

Neste momento, meu conselho é que você descanse um pouco. Este capítulo, apesar de não muito longo, envolveu conceitos não tão simples. Pense um pouco sobre eles e faça uns testes, tente ver onde esse tipo de solução poderia se encaixar no seu atual trabalho.

Capítulo 9

Melhorando performance com Cache

Sempre que entramos no site da Casa do Código, ela nos mostra um monte de livros que podemos comprar. O leitor mais atento já deve ter percebido que essa listagem não muda muito dentro de um determinado intervalo de tempo. E o fato de não mudar nos leva a um questionamento. Será que realmente precisamos ficar executando a lógica do método do controller em todas as vezes que um usuário acessar essa página?

9.1 Cacheando o retorno dos métodos dos controllers

Um dos pontos onde mais podemos ganhar em tempo de execução dentro de uma aplicação é justamente quando não necessitamos realizar a mesma lógica diversas vezes. Pensando em toda a arquitetura de uma aplicação web padrão, existem muitos lugares onde isso é possível.

- Evitar acesso contínuo ao banco de dados;
- Evitar que o navegador fique realizando requisições para arquivos estáticos;
- Evitar que um método do controller, cujo retorno não muda muito, fique sendo executado diversas vezes.

Podemos atacar cada um dos pontos citados aqui, e olha que esses são os mais evidentes. Esse tipo de estratégia, onde guardamos um resultado e ficamos reaproveitando, é comumente conhecido como **Cache**. Como já estamos acostumados, o Spring MVC também fornece ajuda nessa área.

A ideia aqui é: queremos guardar o retorno do método, do controller responsável por retornar o `ModelAndView` relativo ao endereço de exibição dos livros cadastrados. Só para relembrar, dê uma olhada no código:

```
@Controller
@Transactional
@RequestMapping("/produtos")
public class ProductsController {

    @RequestMapping(method=RequestMethod.GET
    public ModelAndView list(){
        System.out.println("listando");
        ModelAndView modelAndView =
            new ModelAndView("products/list");
        modelAndView.addObject("products", products.list());
        return modelAndView;
    }
}
```

Sempre que o usuário acessa o endereço /produtos, o Spring MVC invoca esse método onde acessamos o banco e carregamos os livros cadastrados. Caso nossa página fosse inteiramente fiel à Casa do Código, teríamos ainda mais uma consulta, que retornaria os livros que são lançamentos. O grande ponto é que essas informações não mudam a cada acesso, logo não faz muito sentido ficarmos executando essa lógica o tempo inteiro.

Para resolvermos isso, o Spring MVC provê uma solução muito simples.

```
@RequestMapping(method=RequestMethod.GET)
@Cacheable
public ModelAndView list(){
    ModelAndView modelAndView =
        new ModelAndView("products/list");
    modelAndView.addObject("products", products.list());
    return modelAndView;
}
```

Apenas colocar a annotation @Cacheable em cima do método faz com o que o Spring MVC saiba que, uma vez que o código for executado, ele deve guardar o retorno e utilizá-lo para todas as próximas requisições que caiam no mesmo lugar. Agora já podemos tentar fazer um teste, para verificar se realmente o nosso método só está sendo executado da primeira vez. Só que quando tentamos levantar o servidor, aparece a seguinte exception:

```
java.lang.IllegalStateException: No cache names could be detected
on 'public org.springframework.web.servlet.ModelAndView
br.com.casadocodigo.loja.controllers.ProductsController.list()'
```

Essa exception não é muito clara. O módulo de cache do Spring pede que você defina nomes que podem agrupar um ou mais objetos que você decida colocar no cache. Por exemplo, todos os livros que ficarem no cache podem ficar agrupados sob a chave books. Isso também é conhecido como **cache region**.

```
@RequestMapping(method=RequestMethod.GET)
@Cacheable(value="books")
public ModelAndView list(){
    ModelAndView modelAndView =
```

```
        new ModelAndView("products/list");
    modelAndView.addObject("products", products.list());
    return modelAndView;
}
```

Essa é uma parte importante, quase sempre quando usarmos o `@Cacheable`, vai ser necessário definir o valor para a região de cache.

Pronto, agora podemos subir o servidor normalmente e, além disso, precisamos verificar se o cache está funcionando. Para fazer essa checagem, podemos acessar a tela de produtos mais de uma vez. O normal seria que a *query* realizada pelo Hibernate só fosse mostrada da primeira vez, indicando que o método `list` do nosso controller só executou no primeiro acesso. Só lembrando, as queries estão exibidas porque fizemos essa configuração na classe `JPAConfiguration`.

Caso você tenha realizado o teste, deve ter percebido que o método ainda continua sendo executado todas as vezes.

Habilitando e configurando um provedor de Cache

Vamos precisar habilitar o uso do cache, para que o Spring possa começar a guardar os retornos dos locais indicados com a annotation `@Cacheable`.

```
@EnableWebMvc
@ComponentScan(basePackageClasses={HomeController.class,
    ProductDAO.class,FileSaver.class,ShoppingCart.class})
@EnableCaching
public class AppWebConfiguration {
    ...
}
```

Esse é justamente o papel da annotation `@EnableCaching`. Agora, depois de habilitado, no momento do start do servidor, outra exception é lançada.

```
java.lang.IllegalStateException: No bean of type CacheManager
could be found.Register a CacheManager bean or remove the
EnableCaching annotation from yourconfiguration.
```

Perceba que ela nos informa que o Spring não foi capaz de encontrar nenhuma classe que seja responsável por efetivamente guardar os objetos que devem ser cacheados. A referência à interface `CacheManager` é importante, pois sempre que quisermos usar uma implementação que faça o papel do cache para a gente, teremos que buscar classes que implementem essa interface.

```
...
public class AppWebConfiguration {
    @Bean
    public CacheManager cacheManager(){
        return new ConcurrentMapCacheManager();
    }
}
```

Nesse código, fizemos uso da implementação mais simples, que já vem pronta dentro do próprio Spring, a `ConcurrentMapCacheManager`. Nesse momento, ela é mais do que o suficiente para o que necessitamos. Caso não seja suficiente para o seu projeto, dê um pulo até o final do capítulo e já veja como podemos usar uma segunda opção. Com essas configurações, nossa aplicação já deve subir e conseguir guardar os objetos em cache!

9.2 E quando tiverem novos livros?

Estamos mantendo no cache o retorno do método que lista os livros na Casa do Código. E quando fizermos o cadastro de um novo livro? O que deve acontecer com a listagem? Esse é um problema clássico do cache, em algum momento precisamos renová-lo. Usando a API de cache provida pelo Spring, não poderia ser mais fácil.

```
@RequestMapping(method=RequestMethod.POST)
@CacheEvict(value="books", allEntries=true)
public ModelAndView save(...){
    ...
}
```

A annotation `@CacheEvict` tem justamente essa intenção. Podemos configurar que, quando algum método for invocado, determinada região do

cache deve ter seus valores invalidados. Usamos o atributo `allEntries` para indicar que queremos que todos os valores sejam retirados. Essa annotation suporta configurações mais específicas, entretanto, invalidar todos os valores geralmente vai ser mais do que o suficiente para a sua aplicação. Vale lembrar que o atributo `value` pode receber um array de nomes. Isso seria especialmente útil se tivéssemos cacheado o retorno do método que carrega um produto e o tivéssemos colocado em outra região, por exemplo.

9.3 Usando um provedor de cache mais avançado

O `ConcurrentMapCacheManager`, como já foi citado, é uma implementação de cache muito simples. Vários fatores são importantes, quando vamos colocar objetos no cache.

- Qual o tempo máximo?
- Qual o número limite de objetos?
- Qual o número limite de objetos de determinado tipo?
- Qual o tempo máximo inativo?

Claro que a resposta dessas questões dependem da complexidade do problema que você está querendo resolver. Para tentar cobrir algumas dessas questões, podemos usar uma implementação provida pelo Guava, uma biblioteca criada pelo Google com várias classes que podem ser úteis em qualquer projeto. Siga este link: https://code.google.com/p/guava-libraries/wiki/GuavaExplained para mais detalhes sobre o que é oferecido.

Para resolver o nosso problema, vamos ficar apenas com a parte de cache, fornecida pelo Guava.

```
@Bean
public CacheManager cacheManager() {
    CacheBuilder<Object, Object> builder =
        CacheBuilder.newBuilder()
            .maximumSize(100)
```

```
            .expireAfterAccess(5, TimeUnit.MINUTES);
    GuavaCacheManager cacheManager = new GuavaCacheManager();
    cacheManager.setCacheBuilder(builder);
    return cacheManager;
}
```

A classe `CacheBuilder`, do próprio Guava, serve para criarmos nossa configuração de cache. Perceba que conseguimos definir alguns valores que respondem a algumas das perguntas que fizemos anteriormente. Além disso, o Spring já criou uma extensão que possui uma implementação da interface `CacheManager` justamente para suportar o Guava, no caso a `GuavaCacheManager`. Com essa mudança, saímos de um provedor bem simples de cache para um já mais completo e que, provavelmente, você vai poder usar em algum de seus projetos.

Apenas um último detalhe em relação à utilização do provedor do Guava: precisamos alterar nosso arquivo `pom.xml` para adicionar a extensão do Spring.

```
<!-- cache -->

<dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>18.0</version>
</dependency>

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context-support</artifactId>
    <version>4.1.0.RELEASE</version>
</dependency>
```

Além das opções citadas aqui no livro, o Spring também suporta outros provedores de cache, já prontos.

- EhCache
- GemFire

- JSR-107, especificação do Java para padronizar as APIs de cache.

Caso nenhum desses sirva para você, ainda existe a possibilidade de você implementar a interface `CacheManager` e plugar a sua solução.

9.4 Conclusão

Neste capítulo, vimos uma das soluções mais importantes quando estamos falando de performance de uma aplicação. Evitar o acesso ao disco ou a algum serviço remoto, pode fazer com que sua aplicação diminua drasticamente o tempo de resposta das requisições. O único ponto a que você deve ficar muito atento é em relação a quais objetos devem ir para o cache. Lembre-se sempre, quando um objeto está cacheado, você deve aceitar que seus usuários, nem que seja por um curto período de tempo, podem ver dados que não são os mais atualizados.

Capítulo 10

Respondendo mais de um formato

Nossa aplicação, até o presente momento, só lida com requisições vindas de um navegador, só que agora vamos um pouco além do que já existe na Casa do Código. Sites de vendas muito grandes, como a Amazon e o Submarino, além de exporem os seus produtos em seus sites, fazem parcerias com outros sites para que essas outras aplicações também possam exibir seus produtos.

10.1 Expondo os dados em outros formatos

Quando falamos de integração com outras aplicações, o primeiro ponto em que temos que pensar é: qual é o formato que vamos usar para realizar a integração? Atualmente, nossa aplicação só é capaz de retornar páginas para os

clientes, no caso os navegadores, em HTML. Um outro tipo de cliente, hoje já muito comum, são os celulares com Android ou iOS. E como você já deve esperar, exibir dados através de HTML pode não ser o formato ideal para ser usado nesses aparelhos.

A primeira tarefa que precisamos fazer é ter um outro método no nosso controller, que seja capaz de retornar a lista de livros em outro formato. O escolhido aqui vai ser o JSON, que é um formato muito comum no mercado.

```
...
public class ProductsController {
    @RequestMapping(method = RequestMethod.GET,value="json")
    @ResponseBody
    public List<Product> listJson() {
        return products.list();
    }
}
```

Perceba que só adicionamos mais um método à classe ProductsController, a única diferença foi que usamos a annotation @ResponseBody. Ela informa para o Spring MVC que o retorno do método é para ser usado diretamente como corpo da resposta. Caso você não use essa annotation, ele vai procurar por uma página, que é o comportamento padrão. Basta que alguém acesse o endereço /produtos/json, que o retorno já vai ser um JSON.

Talvez o leitor mais curioso esteja se perguntando: por que o retorno vai ser um JSON, onde foi que configuramos isso? Como já adicionamos a dependência do Jackson no nosso classpath, o Spring MVC já entende que esse é o único formato, além do HTML, cuja configuração já temos.

E se tivermos clientes que trabalhem melhor com XML do que com JSON? Como vamos fazer? Nesse momento, temos que ter um método para cada formato de resposta que precisamos.

10.2 Content negotiation e outros ViewResolvers

Ter um método para cada formato de resposta diferente até funciona, o único problema é que você vai acabar com códigos repetidos. Lembre-se que o único ponto que vai mudar é a representação do retorno, a lógica para recuperar o dado vai ser a mesma. Vamos tentar resolver este problema. Primeiro, precisamos apagar aquele método novo e deixar apenas o antigo, que já adiciona a lista de produtos no `ModelAndView`.

```
@Controller
@Transactional
@RequestMapping("/produtos")
public class ProductsController {
    //apenas lembrando do nosso método que já existe.

    @RequestMapping(method=RequestMethod.GET)
    @Cacheable(value="lastProducts")
    public ModelAndView list(){
        ModelAndView modelAndView =
            new ModelAndView("products/list");
        modelAndView.addObject("products", products.list());
        return modelAndView;
    }
}
```

O ponto agora é: quando um cliente fizer o request para a URL `/produtos`, qual formato vamos retornar? Essa é a parte interessante, o protocolo HTTP já fornece um jeito de lidar com esse problema! Ele permite que o cliente indique qual o formato de resposta que ele prefere. Segue um exemplo de requisição usando a ferramenta **CURL**, famosa no mundo Unix.

```
curl -H "Accept:application/json" -X GET
    "http://localhost:8080/casadocodigo/produtos"

curl -H "Accept:text/html" -X GET
    "http://localhost:8080/casadocodigo/produtos"
```

No primeiro exemplo, fizemos uma requisição indicando que desejamos o retorno no formato `application/json`. Isso é feito através do cabeçalho `Accept`. Perceba que no segundo já indicamos que queremos `HTML` como formato. Essa técnica é conhecida como **Content Negotiation** e é muito utilizada em integrações baseadas no HTTP, também conhecida como REST.

A parte bem legal é que o Spring MVC já oferece esse suporte para nós. Precisamos apenas ensiná-lo que agora ele tem que decidir qual formato retornar baseado no `Accept`.

ContentNegotiatingViewResolver

Para ensinar o Spring MVC sobre qual formato ele deve retornar, vamos usar um novo tipo de *ViewResolver*, o `ContentNegotiatingViewResolver`.

```
...
public class AppWebConfiguration {
    ...
    @Bean
    public ViewResolver contentNegotiatingViewResolver(
            ContentNegotiationManager manager) {
        List<ViewResolver> resolvers =
            new ArrayList<ViewResolver>();

        resolvers.add(internalResourceViewResolver());
        resolvers.add(new JsonViewResolver());

        ContentNegotiatingViewResolver resolver =
            new ContentNegotiatingViewResolver();
        resolver.setViewResolvers(resolvers);
        resolver.setContentNegotiationManager(manager);
        return resolver;
    }
}
```

Perceba que criamos uma lista com implementações de `ViewResolver` diferentes e depois criamos o objeto do tipo `ContentNegotiatingViewResolver` e passamos a lista. Basica-

mente, você pode criar o `ViewResolver` que você quiser, mas vamos ficar na realidade, retornar `HTML`, `JSON` e `XML` já vai ser suficiente, quase sempre.

Outro detalhe importante é que, para lidar com páginas normais, aproveitamos o método `internalResourceViewResolver` que já retorna o objeto responsável pelas JSPs. Já o responsável por tratar o `JSON` vamos ter que criar, pois o Spring MVC não disponibiliza um por padrão.

```
package br.com.casadocodigo.loja.viewresolver;

...

public class JsonViewResolver implements ViewResolver {


    @Override
    public View resolveViewName(String viewName, Locale locale)
    throws      Exception {
        MappingJackson2JsonView view =
            new MappingJackson2JsonView();
        view.setPrettyPrint(true);
        return view;
    }
}
```

O importante é que devemos retornar um objeto do tipo `View`, que efetivamente será responsável por escrever a resposta no cliente. No nosso caso, estamos aproveitando a classe `MappingJackson2JsonView`, que já está disponível no projeto, desde que adicionamos as dependências do Jackson, no capítulo anterior. Como não vamos direcionar para nenhum arquivo, não é necessário o uso do parâmetro que indica o nome da view. Apenas como informação, existe um post no blog do Spring sobre este mesmo assunto, basta seguir o link: http://spring.io/blog/2013/06/03/content-negotiation-using-views/.

Pronto, agora o nosso mesmo método é capaz de retornar formatos diferentes em função do tipo. Para facilitar ainda mais, o Spring MVC já suporta que você defina o formato que você quer através da própria URL. Caso você queira forçar o retorno em `JSON` pelo navegador, basta que acesse a URL

com a extensão `.json`. Por exemplo, http://localhost:8080/casadocodigo/produtos.json. O mesmo vale para `HTML` e outros formatos que você queira suportar.

10.3 Curiosidade sobre o objeto a ser serializado

O nosso método de listagem retorna um objeto do tipo `ModelAndView`, dentro do qual podemos adicionar quantos objetos quisermos para que sejam usados na view. Como que o serializador de `JSON` sabe qual destes objetos escolher?

A maneira padrão é iterar por todas as chaves do `ModelAndView` e gerar o `JSON` para cada um dos objetos encontrados. Caso você queira mudar isso, especificando uma determinada chave, é possível invocar um método de configuração.

```
@Override
public View resolveViewName(String viewName, Locale locale)
    throws Exception {
    MappingJackson2JsonView view = new MappingJackson2JsonView();
    view.setPrettyPrint(true);
    //define a chave
    view.setModelKey("products");
    return view;
}
```

Dessa forma, estamos forçando o `MappingJackson2JsonView` a ignorar as outras chaves. É importante notar que esse comportamento de analisar todos os objetos do `ModelAndView` não é o padrão. O serializador de `XML`, baseado na especificação do JAX-B, busca pelo primeiro objeto compatível!

10.4 Conclusão

Neste capítulo, foi abordado um tema que está em evidência: integração de sistemas via REST. Suportar o Content Negotiation é fundamental para você dar flexibilidade às aplicações clientes sobre qual formato elas preferem. Um

outro ponto importante foi o uso da annotation `@ResponseBody`, ela é muito útil quando você quiser que o retorno do método já seja o corpo da resposta. Só não ficamos com ela porque queríamos suportar o retorno em `HTML` e, para isso, tivemos que user o `ModelAndView` para indicar a página que deveria ser usada.

Uma annotation que não abordamos, mas que também pode ser útil, é `@RestController`. Basta você colocar em cima do seu controller sempre que tiver uma classe onde todos os métodos devem usar o retorno diretamente como corpo da resposta. Caso o nosso controller só retornasse `JSON` ou `XML`, podíamos ter feito isso e nos poupado de escrever `@ResponseBody` em todos métodos.

Capítulo 11

Protegendo a aplicação

Até agora, todas as URLs do nosso sistema estão acessíveis por todos os usuários. Algumas até são liberadas, como a que leva para página inicial, que deve exibir todos os livros. Só que temos algumas URLs que necessitam de um usuário logado, como a que cadastra os novos livros e a de checkout.

Podemos até implementar toda essa parte de segurança na mão, mas como já vem sendo feito no decorrer do livro, vamos usar um módulo do Spring que implementa boa parte do que a gente precisa. É bom sempre lembrar que implementar uma estratégia de segurança não é trivial. Além de forçar o login para algumas URLs, é necessário ter preocupação com pelo menos mais alguns itens.

- Quais perfis podem acessar as URLs, também conhecido como autorização;
- URLs acessadas por vários perfis, mas com trechos de página restritos;

- Fontes de dados diferentes para realização de login.

11.1 Configurando o Spring Security

Para conseguirmos usar as classes do **Spring Security**, precisamos adicionar mais algumas dependências ao nosso projeto. Por isso, vamos alterar o arquivo `pom.xml` de novo.

```
...
<dependencies>
    <!-- Spring security -->

    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-config</artifactId>
        <version>4.0.0.M2</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-taglibs</artifactId>
        <version>4.0.0.M2</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-web</artifactId>
        <version>4.0.0.M2</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-core</artifactId>
        <version>4.0.0.M2</version>
    </dependency>
</dependencies>

<repositories>
    <repository>
        <id>spring-milestones</id>
        <name>Spring Milestones</name>
        <url>http://repo.spring.io/milestone</url>
```

```
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
    </repository>
</repositories>
```

A versão estável do Spring Security ainda é a 3, mas já vamos usar a 4 para garantir compatibilidade com os outros módulos que já estamos utilizando. Como a versão 4 ainda está em fase de construção, temos apenas acesso aos **milestones**. Por conta disso, fomos obrigados a adicionar mais um repositório, para que o Maven consiga buscar todos os `jars` necessários. Como o nosso projeto foi importado como **Maven project**, o *classpath* já é atualizado em função de cada alteração que fazemos no arquivo. Caso você tenha importado como projeto normal, lembre-se sempre de fazer um `mvn eclipse:eclipse`.

Configuração do filtro

Toda lógica de segurança do Spring Security é iniciada na classe `FilterSecurityInterceptor`, que é um filtro da especificação de Servlets. Precisamos apenas configurar para que este filtro seja carregado.

```
package br.com.casadocodigo.loja.conf;

//imports

public class SpringSecurityFilterConfiguration
    extends AbstractSecurityWebApplicationInitializer{

}
```

Aqui usamos a mesma estratégia da classe `ServletSpringMVC`. As duas, no fim, implementam a interface `WebApplicationInitializer`, que é carregada pelo Spring para fazer o registro dos Servlets, filtros e tudo relativo à especificação de Servlets. Caso não se lembre, já comentamos sobre ela no segundo capítulo do livro.

Só que, em vez de implementar diretamente a interface, herdamos da classe `AbstractSecurityWebApplicationInitializer`, que já vem

pronta no Spring Security e faz todo o trabalho de registro para a gente. Pronto, o filtro já está configurado!

11.2 Garantindo a autenticação

Caso você tenha feito a configuração inicial e já tente navegar pela aplicação, vai perceber que todas as URLs ainda estão liberadas. O detalhe é que o filtro já está lá, mas não ensinamos nada para ele ainda.

```
package br.com.casadocodigo.loja.conf;

//imports

@EnableWebSecurity
public class SecurityConfiguration
    extends WebSecurityConfigurerAdapter{

}
```

Aqui herdamos da classe `WebSecurityConfigurerAdapter`, que já fornece toda a infraestrutura pronta para começarmos a fazer nossas configurações de segurança. Além disso, utilizamos a annotation `@EnableWebSecurity`, que deve ser colocada em cima das classes de configuração do Spring Security responsáveis por efetivamente controlar as regras de acesso. Essa annotation faz com que outros componentes sejam carregados. Seguem alguns exemplos.

- **SecurityExpressionHandler**, responsável por avaliar a linguagem específica de controle de segurança, ainda vamos passar por ela.
- **WebInvocationPrivilegeEvaluator**, responsável por verificar se os usuários tem acessos aos endereços.

No fim, podemos fazer tudo isso na mão. O problema é que gastaríamos tempo configurando enquanto poderíamos estar resolvendo o problema da aplicação, que é controlar o acesso. Um último ponto necessário é informar que essa classe deve ser carregada quando nossa aplicação for iniciada. Vamos fazer isso na classe `ServletSpringMVC`.

```
@Override
protected Class<?>[] getRootConfigClasses() {
    return new Class[]{SecurityConfiguration.class};
}

@Override
protected Class<?>[] getServletConfigClasses() {
    return new Class[]{AppWebConfiguration.class,
        JPAConfiguration.class};
}
```

Antes tínhamos usado o método `getServletConfigClasses` e agora fomos obrigados a usar o método `getRootConfigClasses`. O problema é que o filtro do Spring Security é carregado antes do Servlet do Spring MVC, logo, precisamos que os objetos de configuração relativos a ele estejam disponíveis antes. É justamente para isso que serve o `getRootConfigClasses`, ele faz com que as classes sejam lidas e carregadas dentro de um *Listener* que é lido quando o servidor sobe. No caso do Spring MVC, essa classe é a `ContextLoaderListener`.

Agora, sem fazer mais nada, já adicionamos a necessidade de autenticação a todas as URLs. Caso algum usuário tente acessar qualquer endereço, ele vai ser redirecionado para uma tela de login. O único problema é: onde foi que criamos essas regras e de onde veio essa tela de login?

Customizando as regras de acesso

Como a gente não fez nada, o Spring Security usou o método `configure`, que já existe na classe `WebSecurityConfigurerAdapter`, para saber quais eram as nossas configurações de controle de acesso.

```
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .anyRequest().authenticated()
        .and()
        .formLogin().and()
        .httpBasic();
}
```

O objeto do tipo `HttpSecurity` é o ponto de entrada para que possamos customizar as regras de autenticação e autorização. Existem vários métodos sendo invocados nesse trecho de código, vamos só entender o fluxo um pouco mais. O método `authorizeRequests()` é o que nos retorna o objeto onde podemos configurar as regras de acesso em si. Logo após a sua invocação, dizemos que todo request (`anyRequest`) tem que ser feito por alguém autenticado (`authenticated`). O método `formLogin` indica que queremos que nosso sistema suporte autenticação baseada em um formulário comum. O último método, o `httpBasic`, indica que suportamos também o modelo de autenticação, de mesmo nome, provido pelo próprio protocolo HTTP. Fique atento ao método `and()`, ele serve para irmos voltando ao objeto do tipo `HttpSecurity`, para que adicionemos as configurações necessárias. Ainda vamos mexer bastante nessa configuração, para darmos suporte a todas as nossas necessidades.

Essa implementação é a que vem por default no Spring Security, como ela obriga todo request a ser autenticado, não conseguimos nem mais acessar a listagem de livros.

Agora, antes de mais nada, vamos colocar as configurações necessárias de autenticação e autorização para nossos usuários.

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
    .antMatchers("/produtos/form").hasRole("ADMIN")
    .antMatchers("/shopping/**").permitAll()
    .antMatchers(HttpMethod.POST,"/produtos").hasRole("ADMIN")
    .antMatchers("/produtos/**").permitAll()
    .anyRequest().authenticated()
    .and().formLogin();
}
```

O código é quase que autoexplicativo. Vamos pegar alguns trechos para dar uma olhada.

```
antMatchers("/produtos/form").hasRole("ADMIN")
```

Estamos dizendo que, além de estar logado, o usuário precisa ser **ADMIN** para acessar este endereço da nossa aplicação. Um outro caso interessante:

```
.antMatchers(HttpMethod.POST,"/produtos").hasRole("ADMIN")
```

Caso o endereço `/produtos` seja acessado através do verbo `POST`, só liberamos acesso caso tenha sido disparado por um usuário com perfil **ADMIN**. Um último que vale a observação:

```
.antMatchers("/produtos/**").permitAll()
```

Aqui estamos dizendo que todos os outros endereços que comecem com `/produtos` estão liberados. O método `antMatcher` recebe uma expressão regular no mesmo estilo suportado pela ferramenta `ANT`, famosa no mundo Java na parte de builds.

Após as configurações de URLs, informamos que todo o resto só está liberado caso o usuário esteja no mínimo autenticado. Fique atento para a ordem das invocações, ela é importante! Primeiro, faça as restrições e, depois, libere todo o resto. O Spring Security verifica as restrições na ordem em que elas foram cadastradas. Caso você adicione a regra que bloqueia tudo no início da configuração, todas as URLs ficarão bloqueadas.

11.3 Configuração da fonte de busca dos usuários

Os acessos às nossas URLs já estão devidamente configurados, entretanto, ninguém ainda consegue fazer o login na nossa aplicação. Precisamos ensinar ao Spring Security de onde ele deve buscar os usuários para aplicar as nossas regras. Visando permitir implementar a busca da melhor maneira para cada aplicação, eles disponibilizaram uma interface chamada `UserDetailsService`. Vamos dar uma olhada em como vai ficar nossa classe de configuração depois de termos implementado a interface.

```
@EnableWebSecurity
public class SecurityConfiguration
    extends WebSecurityConfigurerAdapter{
    ....

@Autowired
```

```
private UserDetailsService users;

@Override
protected void configure(AuthenticationManagerBuilder auth)
        throws Exception {
    auth.userDetailsService(users).
        passwordEncoder(new BCryptPasswordEncoder());
}
```

Perceba que vamos receber injetada justamente a implementação da interface que ainda vamos criar. Depois disso, usamos uma sobrecarga do método **configure** que recebe um `AuthenticationManagerBuilder`, objeto que nos permite associar um novo `UserDetailsService` ao Spring Security. Além disso, ainda forçamos que a senha seja mantida utilizando um algoritmo de hash; escolhemos o BCrypt. Poderiam existir vários capítulos somente dedicados a algoritmos de hash, já que estes formam uma vasta área de estudo. Aqui no livro, vamos nos restringir somente a usar o BCrypt, que a própria documentação do Spring Security estimula que seja usado.

Criando o nosso UserDetailsService

No caso do nosso sistema, para recuperar um usuário através do login, vamos simplesmente criar um DAO.

```
package br.com.casadocodigo.loja.daos;

//imports

@Repository
public class UserDAO implements UserDetailsService{

    @PersistenceContext
    private EntityManager em;

    @Override
    public UserDetails loadUserByUsername(String username)
            throws UsernameNotFoundException {
        String jpql = "select u from User u
```

```
                    where u.login = :login";

        List<User> users = em.createQuery(jpql,User.class)
            .setParameter("login", username).getResultList();

        if(users.isEmpty()){
            throw new UsernameNotFoundException
                ("O usuario "+username+" não existe");
        }
        return users.get(0);
    }

}
```

"

Nosso método é bem comum: simplesmente realizamos uma query e devemos retornar um usuário encontrado, ou uma exception indicando o contrário. Perceba que o nome do método é `loadUserByUsername`, justamente o que a interface nos obriga implementar. Uma outra questão interessante é que ele não recebe a senha como argumento, por que será? O principal motivo é para você não ter que lidar com o processo de aplicar o hash na senha antes de fazer a consulta. Você simplesmente busca pelo login e o Spring Security vai verificar se a senha cadastrada bate com a senha com que foi enviada pelo formulário.

Ainda em relação à assinatura do método `loadUserByUsername`, outro ponto que chama a atenção é o retorno. Ele nos força a retornar um objeto que implementa a interface `UserDetails`. E, se pensarmos bem, faz todo o sentido. Com o objeto que representa o usuário logado na mão, como o Spring Security ia saber qual era a senha dele? E quais perfis estão associados a ele? É para isso que serve a interface: para definir os métodos que ele poderá invocar e recuperar estas informações.

Para satisfazê-lo, vamos criar nossa classe `User` e implementar a interface exigida.

```
package br.com.casadocodigo.loja.models;

//outros imports
```

```
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;

@Entity
public class User implements UserDetails{

    @Id
    private String login;
    private String password;
    private String name;
    @ManyToMany(fetch = FetchType.EAGER)
    private List<Role> roles = new ArrayList<>();

    //outros getters e setters

    @Override
    public String getPassword() {
        return password;
    }

    @Override
    public Collection<? extends GrantedAuthority>
    getAuthorities() {
        return roles;
    }

    @Override
    public String getUsername() {
        return login;
    }

    @Override
    public boolean isAccountNonExpired() {
        return true;
    }

    @Override
    public boolean isAccountNonLocked() {
```

```
        return true;
    }

    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }

    @Override
    public boolean isEnabled() {
        return true;
    }

}
```

Fomos obrigados a implementar até mais métodos do que estávamos esperando, vamos dar uma olhada nos mais importantes para a gente.

- `getPassword`: retorna o password do usuário;
- `getUsername`: retorna o login do usuário;
- `getAuthorities`: retorna os perfis do usuário autenticado.

Os outros métodos são para um controle mais fino. O importante é: todos os métodos que retornam *boolean* devem retornar `true` para garantir o acesso do usuário às URLs protegidas. Caso o seu sistema precise controlar renovação de senha ou, por exemplo, bloqueio por inatividade, estes outros métodos podem ser úteis.

Perceba que o método `getAuthorities` deve retornar uma lista de objetos que implementam a interface `GrantedAuthority`. Essa é justamente a interface que nossa classe que representa o perfil deve implementar.

```
package br.com.casadocodigo.loja.models;

//outros imports
import org.springframework.security.core.GrantedAuthority;
```

```
@Entity
public class Role implements GrantedAuthority {

    @Id
    private String name;

    @Override
    public String getAuthority() {
        return name;
    }
```

Pronto, com ambas as classes implementando a interface e sendo entidades gerenciadas pela JPA, já podemos realizar efetivamente o login dos usuários. Apenas para facilitar, você pode executar os inserts a seguir para cadastrar usuários e seus perfis.

```
insert into Role values('ROLE_ADMIN');
insert into Role values('ROLE_COMPRADOR');

insert into User(login,name,password)
    values('comprador@gmail.com',
    'Alberto',

                                                    );
insert into User(login,name,password)
    values('admin@casadocodigo.com.br'
    ,'Administrador',

                                                    );


insert into User_Role(User_login,roles_name)
    values('comprador@gmail.com','ROLE_COMPRADOR');
insert into User_Role(User_login,roles_name)
    values('admin@casadocodigo.com.br','ROLE_ADMIN');
```

Provavelmente, no seu sistema, você vai precisar cadastrar novos usuários através de um formulário. Para aplicar o hash na senha passada pelo formulário de cadastro, lembre-se de usar a classe `BCryptPasswordEncoder`.

Problema com a ordem de carregamento das configurações

Caso você tente subir a aplicação, vai receber a seguinte exception:

```
Caused by: org.springframework.beans.factory
              .NoSuchBeanDefinitionException:
No qualifying bean of type
[org.springframework.security.core.userdetails
    .UserDetailsService]
```

Ela informa que não conseguiu achar nenhuma implementação para a interface `UserDetailsService`. O problema é que o DAO está configurado para ser carregado na classe `AppWebConfiguration` e ela só é carregada no momento de carregamento do Servlet do Spring MVC.

```
@Override
protected Class<?>[] getServletConfigClasses() {
    return new Class[]{AppWebConfiguration.class,
    JPAConfiguration.class};
}
```

Precisamos que ela passe para o método `getRootConfigClasses` para ser carregada no inicio da aplicação e já fazer com que os objetos fiquem disponíveis para serem lidos pelo filtro.

```
@Override
protected Class<?>[] getRootConfigClasses() {
    return new Class[]{SecurityConfiguration.class,
                       AppWebConfiguration.class};
}

@Override
protected Class<?>[] getServletConfigClasses() {
    return new Class[]{JPAConfiguration.class};
}
```

Caso tente subir o servidor de novo, vai ver que felizmente o erro vai mudar.

```
Error creating bean with name 'userDAO': Injection of persistence
dependencies failed; nested exception is
org.springframework.beans.factory.NoSuchBeanDefinitionException:
No qualifying bean of type
[javax.persistence.EntityManagerFactory] is defined
```

Ele não consegue criar o objeto do tipo `UserDAO`, já que não sabe como criar uma `EntittyManagerFactory`. É o mesmo problema do caso anterior: para criar o DAO, é necessária a criação do `EntityManager`, e o mesmo só é carregado no momento de criação do Servlet. Vamos fazer a última alteração nessa classe!

```
@Override
protected Class<?>[] getRootConfigClasses() {
    return new Class[]{SecurityConfiguration.class,
    AppWebConfiguration.class,JPAConfiguration.class};
}

@Override
protected Class<?>[] getServletConfigClasses() {
    return new Class[]{};
}
```

Agora tudo é carregado no início, e com isso resolvemos esse problema chato da ordem de carregamento.

Curioside sobre o prefixo ROLE_

Talvez o leitor mais atento esteja se perguntando o motivo de cadastrarmos os perfis com o prefixo **ROLE_**. Para implementar as verificações de acesso, tanto de usuário autenticado quanto de autorizado, o Spring Security fornece uma interface chamada `AccessDecisionVoter`. A classe padrão, que faz a verificação dos perfis, é a `RoleVoter`.

```
public class RoleVoter implements AccessDecisionVoter<Object> {


    private String rolePrefix = "ROLE_";
```

```
    public boolean supports(...){
        //verifica se deve aplicar verificações
    }
    }
```

Perceba que ela força o uso do prefixo, é apenas uma decisão de projeto. Imagine que você possa criar um `AccessDecisionVoter` customizado e associar outro padrão interno de perfil da sua empresa; como que o `RoleVoter` ia saber se devia aplicar a verificação ou não? É justamente para isso que ele força o uso do prefixo. Você agora deve estar imaginando outra coisa. Lá na configuração inicial, não foi bem assim que fizemos.

```
.antMatchers("/produtos/form").hasRole("ADMIN")
```

Onde está o prefixo? É que o método `hasRole` já adiciona para a gente.

```
Assert.notNull(role, "role cannot be null");
if (role.startsWith("ROLE_")) {
    throw new IllegalArgumentException(
        "role should not start with 'ROLE_'
        since it is automatically inserted.
        Got '" + role + "'");
}
String role = "ROLE_" + role;
...
```

O método `hasRole` só existe por conta do `RoleVoter`.

11.4 Cross-site request forgery

Agora que já conseguimos logar, podemos navegar entre as páginas do sistema. Um problema acontece na hora que tentamos, por exemplo, adicionar um produto ao nosso carrinho de compras. Quando clicamos no botão para realizar a ação, é retornado o status 403, indicando acesso negado. Ainda vem acompanhando da seguinte mensagem:

```
HTTP Status 403 - Invalid CSRF Token 'null' was found on the
request parameter '_csrf' or header 'X-CSRF-TOKEN'.
```

A sigla CSRF significa *Cross-Site Request Forgery* e é um tipo de ataque que pode ser feito contra sua aplicação. A ideia basicamente é que dados possam ser enviados para a nossa aplicação sendo provenientes de uma outra página qualquer, aberta no seu navegador. Como a Casa do Código é bem famosa, algum outro site poderia esconder um formulário na sua página e, quando um usuário que estivesse logado na Casa do Código interagisse com essa página externa, ela poderia submeter dados com um valor mentiroso para o endereço de checkout. Para conseguir mais informações, você pode acessar a página do próprio Spring Security, http://docs.spring.io/spring-security/site/docs/4.0.0.M2/reference/htmlsingle/#csrf.

Como foi falado no inicio do capítulo, segurança vai além de verificar login e perfis, e se proteger de ataques como esse faz parte do pacote. A proteção padrão contra o CSRF é feita através da geração de um número para cada usuário que se logue no sistema. A partir disso, todo `POST` que for feito deve ser acompanhado desse número, também chamado de **token**. Dessa forma, por mais que o site malicioso coloque um formulário escondido, o request não vai ser acompanhado do token, sendo automaticamente banido pela nossa aplicação.

Para a nossa sorte, o Spring Security já provê a implementação para a proteção contra o CSRF. Talvez o leitor tenha percebido que a proteção já está ativada até para requests para URLs que não estão sob proteção. Por padrão, o Spring Security já gera o token a partir do primeiro request do usuário, para a aplicação que está fazendo uso do módulo. A consequência disso é que a verificação do CSRF já é feita para todo `POST`. Para adicionarmos o token, basta que adicionemos o parâmetro com o token nos nossos formulários.

```
<form ...>
 <input type="hidden"
    name="${_csrf.parameterName}"
    value="${_csrf.token}"/>
</form
```

A variável de nome `_csrf` contém a referência para um objeto do tipo `DefaultCsrfToken`. Através dele, temos acesso ao nome do parâmetro e ao valor atual do token. O nome nome da variável pode ser modificado, mas não é necessário nem que escrevamos o input em si. Podemos importar um

conjunto de taglibs do Spring Security que já possui a tag que gera o mesmo input.

```
<%@taglib uri="http://www.springframework.org/security/tags"
    prefix="security" %>

...
<form ...>
    <security:csrfInput/>
</form>
```

A segunda opção é bem melhor, já que não é necessário ficar lembrando do nome da variável e, muito menos, ficar correndo o risco de escrever errado. Caso você não esteja contente com nenhuma das duas, ainda podemos optar por uma terceira.

Anotamos a classe de configuração do Spring Security com a annotation `@EnableWebSecurity`. Só que essa annotation carrega configurações importantes para uma aplicação web, mas não assume que estamos usando o Spring MVC como framework web. Podemos, em vez disso, começar a usar a annotation `@EnableWebMvcSecurity`. Ela carrega os mesmos componentes da anterior e ainda adiciona configurações específicas para o projeto que utiliza o Spring MVC. Uma delas é adicionar o input do CSRF em todos os formulários construídos com a tag `<form:form>`. Nossa classe de configuração ficaria como o exemplo a seguir.

```
@EnableWebMvcSecurity
public class SecurityConfiguration extends
    WebSecurityConfigurerAdapter{
    ...
}
```

Agora só precisamos trocar os nossos formulários. Vamos pegar o de adição de um produto no carrinho, por exemplo. Atualmente, ele está assim:

```
<form action="<c:url value="/shopping"/>" method="post"
    class="container">
    ...
</form>
```

Podemos substituí-lo pela tag de geração de formulário do Spring MVC.

```
<form:form servletRelativeAction="/shopping"
    cssClass="container">
    ....
</form:form>
```

Pronto, agora não precisamos mais nos preocupar em adicionar o input com o token do CSRF. De brinde, ainda não precisamos mais ficar usando a tag `<c:url>`. Quando utilizamos o atributo `servletRelativeAction`, o contexto da aplicação já é adicionado antes da URL! Lembre-se, é uma ótima prática usar as tags de formulário do Spring MVC, geralmente ela já vai fazer boa parte do trabalho padrão para você, como já vimos quando discutimos a validação.

11.5 Customizando mais alguns detalhes

Até agora temos usado uma tela de login que tem sido gerada automaticamente pelo próprio Spring Security. Só que, na nossa aplicação, precisamos ter a opção de criar uma tela com a cara da Casa do Código. E é claro que isso não vai ser um problema, basta ensinarmos ao Spring Security que a URL de login deve ser direcionada para um controller da nossa aplicação.

```
@EnableWebMvcSecurity
public class SecurityConfiguration extends
    WebSecurityConfigurerAdapter{

    @Override
    protected void configure(HttpSecurity http) throws Exception{
        http.authorizeRequests()
        .antMatchers("/produtos/form").hasRole("ADMIN")
        .antMatchers("/shopping/**").permitAll()
        .antMatchers(HttpMethod.POST,"/produtos")
            .hasRole("ADMIN")
        .antMatchers("/produtos/**").permitAll()
        .anyRequest().authenticated()
        .and()
        .formLogin().loginPage("/login").permitAll();
    }
```

O método formLogin() retorna um objeto do tipo LogoutConfigurer e, através dele, conseguimos trocar as configurações que vêm por default para a URL do formulário de login. Esse é o objetivo do método loginPage, ele recebe como argumento a URL que deve ser usada para ir para a tela de login. Devemos invocar o método permitAll para informar que esse endereço está liberado para todos os usuários.

O JSP da tela podia ser parecido com o que segue:

```
<%@ taglib uri="http://www.springframework.org/tags/form"
    prefix="form"%>
    <form:form servletRelativeAction="/login">
            <div>
                <label>
                User
                <input type='text' name='username' value=''>
                </label>
            </div>
            <div>
                <label>Password
                <input type='password' name='password' />
                </label>
            </div>
            <div>
                <input name="submit" type="submit"
                    value="Login" />
            </div>
        </table>
    </form:form>
```

O nome dos parâmetros de login e senha também são definidos através do LoginConfigurer. O default é username para o login e password para a senha. A action do formulário está com o valor /login, que é a URL que, quando feito um POST, dispara o processo de login pelo filtro. O controller que leva para essa página fica por sua conta, leitor!

URL de logout e telas de erro

Para habilitar um endereço que dispara o processo de logout da aplicação basta que invoquemos o método `logout()`.

```
.formLogin().loginPage("/login").permitAll()
.and()
.logout().logoutRequestMatcher(new AntPathRequestMatcher(
                                                    "/logout"));
```

Por default, o Spring Security só libera que o logout seja feito através de `POST`, justamente para forçar a passagem do token do CSRF. Aqui fizemos um pouco diferente, por isso a invocação do método `logoutRequestMatcher`. A URL passada para ele pode ser acessada via `GET` e, mesmo assim, o processo de logout vai ser iniciado.

Outro detalhe importante é a tela de acesso não autorizado. Ela acontece quando o usuário logado não tem o perfil necessário para acessar a URL, é o caso do cadastro de livros, só usuários administradores podem acessar. Quando um usuário tenta acessar uma URL dessas, o Spring Security gera um retorno com o status 403, que significa **Not Authorized**. Para customizar essa tela de erro, basta que façamos uso do arquivo `web.xml` da aplicação.

```
<error-page>
    <error-code>403</error-code>
    <location>/WEB-INF/403.jsp</location>
</error-page>
```

Um detalhe interessante é que até agora não tínhamos precisado desse arquivo. Para quem não se lembra, ele é utilizado para fazermos diversas configurações relativas à nossa aplicação web, mas como quase tudo pode ser configurado via código ou annotation, ele está sendo usado cada vez menos. Para gerar o arquivo pelo Eclise, basta usar a opção **Generate deployment Descriptor stub**.
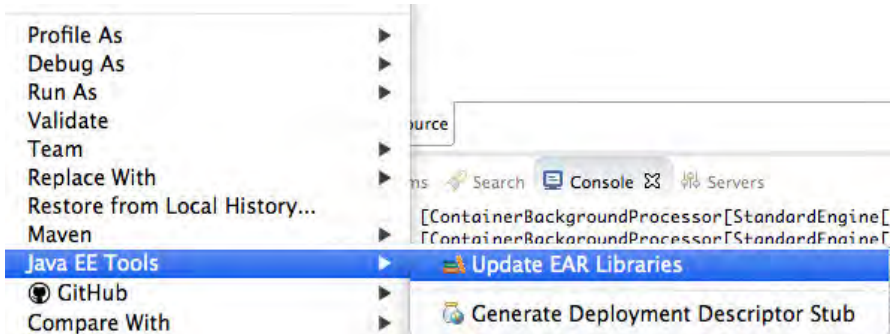
Fig. 11.1: Menu de geração de arquivo web.xml

11.6 Exibindo o usuário logado e escondendo trechos da página

Um detalhe comum é exibir o nome do usuário logado nas páginas que ele acessa dentro do sistema. Podemos fazer isso usando as taglibs disponibilizadas pelo próprio Spring Security. Vamos alterar a página `list.jsp`:

```
<%@ taglib prefix="security"
    uri="http://www.springframework.org/security/tags" %>

<security:authentication property="principal" var="user"/>
Olá ${user.name}
```

A taglib `authentication` acessa o objeto do Spring Security que guarda o usuário que foi carregado pelo `UserDetailsService`. Ele implementa a interface `Authentication`, que possui o método `getPrincipal`. Por isso, passamos o valor `principal` para o atributo `property`. Internamente, a taglib vai buscar pelo `getter` relativo à propriedade. Para podermos usá-lo no resto da página, exportamo-lo para a variável `user`.

Ainda em relação a essa parte, sobrou um detalhe. Essa página pode ser acessada tanto por usuários logados como por qualquer anônimo. Caso aconteça o último caso, neste momento, vamos receber um erro informando que a propriedade `name` não existe. Para resolver isso, vamos exibir esse trecho apenas para usuários logados!

```
<sec:authorize access="isAuthenticated()">
    <sec:authentication property="principal" var="user"/>
    <div>
        Olá ${user.name}
    </div>
</sec:authorize>
```

A tag `authorize` serve justamente para exibir um trecho de página apenas se alguma condição for atendida. Nesse caso, estamos verificando se o usuário está logado. O atributo `access` recebe algumas expressões que são suportadas pelo Spring Security. Foi usada a `isAuthenticated()`, mas podiam ser várias outras. Outra muito comum é `hasRole(...)`, que geralmente é usada quando se deseja restringir certo trecho da página apenas para alguns perfis.

```
<sec:authorize access="hasRole('ROLE_ADMIN')">
    <li><a href="${spring:mvcUrl('PC#form').build()}">
        Cadastrar novo produto</a>
    </li>
</sec:authorize>
```

Para a lista completa, você pode acessar a documentação do Spring Security. Caso precise, siga este link: http://docs.spring.io/spring-security/site/docs/4.0.0.M2/reference/htmlsingle/#el-access.

11.7 Conclusão

Neste capítulo, vimos o suficiente do Spring Security para a maioria das aplicações. Este é um dos melhores módulos do Spring e há ainda várias outras configurações possíveis. Outro ponto importante é que ele é bem extensível, talvez você se lembre das interfaces que foram mostradas durante o capítulo. Caso você precise de algum detalhe específico, pesquise primeiro dentro do framework antes de fazer uma solução caseira.

Como o capítulo foi muito extenso, dê uma pausa para tomar uma água e processar tudo que você leu. Os próximos capítulos, apesar de importantes, são mais simples de serem entendidos.

Capítulo 12

Organização do layout em templates

Já implementamos algumas das funcionalidades que existem dentro do sistema real da Casa do Código. Uma parte com que não nos preocupamos até agora é em relação ao layout da aplicação. Só que até mais importante que o layout é como vamos organizá-lo. Por exemplo, atualmente, entre as páginas que já implementamos, temos a de exibição do detalhe de um livro e a que exibe os produtos do carrinho de compras. Olhando com atenção para elas, vamos perceber que ambas possuem o mesmo header e footer.

```
<header id="layout-header">
    ...
</header>
<nav class="categories-nav">
```

```
    ...
</nav>

<!-- resto da página-->

<footer id="layout-footer">
    ...
</footer>
```

Aqui temos o problema clássico de repetição. Caso alguma mudança seja necessária nesses trechos, vamos ter que sair caçando em todas as páginas. Uma maneira comum de resolver essa situação é utilizando o sistema de *includes* que já existe nas JSPs.

```
<!doctype html>
<!--[if lt IE 7]><html class="no-js lt-ie9 lt-ie8 lt-ie7"
 lang="pt">
<![endif]-->
<!--[if IE 7]>
    <html class="no-js lt-ie9 lt-ie8" lang="pt">
<![endif]-->
<!--[if IE 8]><html class="no-js lt-ie9" lang="pt"><![endif]-->
<!--[if gt IE 8]><!-->
<html class="no-js" lang="pt">
<!--<![endif]-->
<head>
<meta charset="utf-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
<meta name="viewport"
    content="width=device-width, initial-scale=1,
    maximum-scale=1">

<title>${title}</title>
</head>

<body class="${bodyClass}">
<%@include file="/WEB-INF/header.jsp" %>

<!-- resto da página -->
```

```
<%@include file="/WEB-INF/footer.jsp" %>
```

Até isolamos o header e o footer, só que sobrou esse inicio de declaração do HTML. E aí você pode pensar "não tem problema, é só criar mais uma *include* e isso também vai estar isolado". Só que aí é que entra o problema: para uma pessoa nova montar uma página, seguindo o padrão do sistema, ela tem que saber de todas essa *includes* e ainda tem que saber a ordem! A tendência é só piorar.

12.1 Templates

Para amenizar esse problema, a especificação de JSP já pensou em uma solução. A ideia é que possamos criar um modelo de página que possa ser utilizada por todas as outras páginas.

```
<%@taglib tagdir="/WEB-INF/tags" prefix="customTags"%>

<customTags:pageTemplate>
    <section class="container middle">
            <h2 id="cart-title">Seu carrinho de compras</h2>
            <table id="cart-table">
                <colgroup>
                    <col class="item-col">
                    <col class="item-price-col">
                    <col class="item-quantity-col">
                    <col class="line-price-col">
                    <col class="delete-col">
                </colgroup>
                <thead>
                    <tr>
                        <th class="cart-img-col"></th>
                        <th width="65%">Item</th>
                        <th width="10%">Preço</th>
                        <th width="10%">Quantidade</th>
                        <th width="10%">Total</th>
                        <th width="5%"></th>
                    </tr>
```

```
                </thead>
                <tbody>
                ...
</customTags:pageTemplate>
```

Ainda não se preocupe com a tag `masterPage`, vamos resolvê-la daqui a pouco. Vamos focar na estrutura da página. Não precisamos nos preocupar com muita coisa, apenas em adicionar uma tag que indica que queremos usar um modelo de página. A ideia é que essa tag defina o header, footer, outros trechos em comum e apenas receba como parâmetro o corpo da página específica. E se a equipe de front quiser reformular a estrutura da página e mudar a ordem dos menus, onde devemos mudar? Apenas no template!

Perceba que importamos as tags apontando para uma pasta em `WEB-INF`. Diferente do que fizemos no capítulo de validação, vamos criar uma tag quase como se fosse uma JSP, também conhecida como `Tag File`. Nesse caso, vamos chamar a nossa tag de `pageTemplate.tag`. A especificação nos obriga a deixar os arquivos `.tag` dentro da pasta tags.

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core"  prefix="c"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/fmt"  prefix="fmt"%>
<%@ taglib
 uri="http://www.springframework.org/tags/form" prefix="form"%>
<%@taglib uri="http://www.springframework.org/tags"
 prefix="spring" %>
<!doctype html>
<!--[if lt IE 7]>
    <html class="no-js lt-ie9 lt-ie8 lt-ie7" lang="pt">
<![endif]-->
<!--[if IE 7]>
    <html class="no-js lt-ie9 lt-ie8" lang="pt">
<![endif]-->
<!--[if IE 8]><html class="no-js lt-ie9" lang="pt"><![endif]-->
<!--[if gt IE 8]><!-->
<html class="no-js" lang="pt">
<!--<![endif]-->
<head>
<meta charset="utf-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
```

```
<meta name="viewport"
    content="width=device-width, initial-scale=1,
    maximum-scale=1">

<title>Titulo da página</title>
</head>
<body class="classeDoBody">

    <%@include file="/WEB-INF/header.jsp" %>

    <jsp:doBody/>

    <%@include file="/WEB-INF/footer.jsp" %>

</body>
</html>
```

É quase uma JSP, a única diferença é que ela pode ser reaproveitada como uma tag. Perceba que começamos o HTML e usamos as *includes* para deixar a página organizada. A grande jogada está no uso da tag `<jsp:doBody>`, ela que é a responsável por capturar o corpo passado como parâmetro e utilizá-lo dentro da nossa estrutura.

Outro ponto importante é que toda página define um título e uma classe que deve ser usada na tag `body`. Como criamos uma tag, nada nos impede de declarar atributos para ela.

```
<%@attribute name="title" required="true" %>
<%@attribute name="bodyClass" required="true" %>

...

<title>${title}</title>

...

<body class="${bodyClass}">

    <%@include file="/WEB-INF/header.jsp" %>
```

```
    <jsp:doBody/>

    <%@include file="/WEB-INF/footer.jsp" %>
</body>
```

Existem outras bibliotecas que podem ser usadas para a criação de templates, como o Freemarker, Sitemesh, Tiles e Velocity e o Spring MVC tem integração com todas elas. O ponto é que a mesma funcionalidade já é suportada pela JSP e, por esse motivo, este autor prefere ficar com o que já vem pronto na linguagem.

12.2 Deixando o template ainda mais flexível

Um detalhe com o qual é um pouco mais complicado de lidar é que certas páginas precisam de um trecho extra de Javascript logo antes do fechamento do body. Por exemplo, a página de listagem dos itens do carrinho exige que o seguinte trecho seja colocado:

```
<script>
    $(function() {
        $('#checkout').click(function() {
            _gaq.push([ '_trackPageview',
            '/checkout/finalizaCompra' ]);
        });

        $('.book-suggest').click(function() {
            var book = $(this).data('book');
            _gaq.push([ '_trackEvent', 'Recomendação',
            'Livro', book ]);
        });
    });
</script>

<script>
    $(function() {
        $('a[href^="http"]').not('.dont-track')
        .filter(function(index) {
            var ccb = $(this).attr('href')
```

```
            .indexOf("casadocodigo.com.br");
            if (ccb == -1)
                ccb = $(this).attr('href').indexOf("localhost");

            return ccb != 7 && ccb != 11;
        }).click(function(event) {
            var domain = this.href;
            domain = domain.substring(7);
            domain = domain.substring(0, domain.indexOf('/'));

            if (domain.substring(0, 4) == 'www.')
                domain = domain.substring(4);

            _gaq.push([ '_trackPageview', '/LinkExterno/' +
                        this.href ]);
        });
    });
</script>
```

Para contornar esse problema, podemos dizer que nossa tag pode receber fragmentos extras de páginas.

```
<%@attribute name="title" required="true" %>
<%@attribute name="bodyClass" required="true" %>
<%@attribute name="extraScripts" fragment="true" %>

<body class="${bodyClass}">

    <%@include file="/WEB-INF/header.jsp" %>

    <jsp:doBody/>

    <%@include file="/WEB-INF/footer.jsp" %>

    <jsp:invoke fragment="extraScripts"/>
</body>
</html>
```

Agora qualquer página que precise pode passar esse fragmento extra.

```
<customTags:pageTemplate bodyClass="cart" title="${title}">
    <jsp:attribute name="extraScripts">
        ...
    </jsp:attribute>
    <jsp:body>
        <!-- Resto do corpo da página-->
    </jsp:body>

</customTags:pageTemplate>
```

A tag `<jsp:attribute>` é utilizada para indicar o trecho que deve ser usado pela tag `<jsp:invoke>`. Você pode ter quantos fragmentos forem necessários, tudo vai depender da complexidade do seu template. Apenas para completar, alguns detalhes importantes. Quando fazemos uso do `attribute` somos obrigados a usar a tag `<jsp:body>` para passar o corpo, não podemos mais deixá-lo livre. Um último ponto é a ordem: é necessário que o `<jsp:body>` seja a última.

12.3 Conclusão

Este capítulo foi bem rápido, apenas organizamos um pouco mais nosso layout. Não teve nada a ver com o Spring MVC em si. Mesmo assim é importante, já que as telas compõem uma parte fundamental de qualquer aplicação. Não pare agora, já comece o próximo capítulo para que possamos fazer com que nossas telas falem diversas línguas.

Capítulo 13

Internacionalização

Até este momento, todos os textos da aplicação estão escritos diretamente nas páginas. Por exemplo, vamos analisar o menu de navegação da *include* que contém o header do projeto.

```
<ul class="container">
    <li class="category">
        <a href="http://www.casadocodigo.com.br">Home</a>
    <li class="category"><a href="/collections/livros-de-agile">
            Agile </a>
    <li class="category">
        <a href="/collections/livros-de-front-end">Front End</a>
    <li class="category"><a href="/collections/livros-de-games">
            Games </a>
    <li class="category"><a href="/collections/livros-de-java">
            Java </a>
```

```
<li class="category"><a href="/collections/livros-de-mobile">
        Mobile </a>
<li class="category"><a
    href="/collections/livros-desenvolvimento-web"> Web </a>
<li class="category">
    <a href="/collections/outros"> Outros </a>
```

Todos os textos da aplicação estão escritos diretamente na página e, em geral, talvez isso não seja um problema. Entretanto, a Casa do Código já tem a sua versão internacional, a Code Crushing. Elas têm o mesmo layout e só mudam em uma coisa: todos os textos são escritos em inglês.

13.1 Isolando os textos em arquivos de mensagens

Caso continuemos com a estratégia de escrever os textos diretamente na página, teríamos que duplicar cada uma delas. Para conseguirmos **internacionalizar** nossa aplicação, vamos ter que isolar os textos no nosso arquivo `properties`.

```
...

navigation.category.agile = Agilidade
navigation.category.front_end = Front end
navigation.category.games = Jogos
navigation.category.java = Java
navigation.category.mobile = Desenvolvimento móvel
navigation.category.web = Web
navigation.category.others = Outros
```

Mantivemos as chaves que já existiam lá e adicionamos as novas, referentes ao menu. A nomenclatura em um arquivo `properties` sempre é fonte de discussão. Aqui estamos seguindo o mais básico: dividimos os grupos com o `.` e usamos `_` para separar o nome do item em si. Agora que já isolamos os textos no arquivo `messages`, podemos usar as chaves nas nossas páginas e tags.

```
<%@taglib uri="http://java.sun.com/jsp/jstl/fmt"  prefix="fmt"%>
<%@taglib uri="http://www.springframework.org/tags"
    prefix="spring" %>

...
<nav class="categories-nav">
    <ul class="container">
        <li class="category">
            <a href="http://www.casadocodigo.com.br">Home</a>
        <li class="category">
            <a href="/collections/livros-de-agile">
                <fmt:message key="navigation.category.agile"/>
            </a>
        <li class="category">
            <a href="/collections/livros-de-front-end">
                <fmt:message key="navigation.category.front"/>
            </a>
        <li class="category">
            <a href="/collections/livros-de-games">
                <fmt:message key="navigation.category.games"/>
            </a>
        <li class="category">
            <a href="/collections/livros-de-java">
                <fmt:message key="navigation.category.java"/>
            </a>
        <li class="category">
            <a href="/collections/livros-de-mobile">
                <fmt:message key="navigation.category.mobile"/>
            </a>
        <li class="category">
            <a href="/collections/livros-desenvolvimento-web">
                <fmt:message key="navigation.category.web"/>
            </a>
        <li class="category">
            <a href="/collections/outros">
                <fmt:message key="navigation.category.others"/>
            </a>
    </ul>
</nav>
```

Perceba que usamos a tag `<fmt:message>`. O Spring já disponibiliza as chaves no contexto da JSTL para que possamos continuar usando a mesma tag. Outra opção seria usar a tag do próprio Spring.

```
<spring:message code="navigation.category.agile"/>
```

A diferença é que a tag do Spring já tem mais algumas opções, como o valor default e forma mais fácil de passar parâmetros.

13.2 Accept-Language header

Até agora temos apenas o arquivo `messages.properties` com os textos em português. Só que precisamos suportar as mensagens também em inglês. Para fazer isso, precisamos criar o arquivo com a extensão da localização que queremos suportar. Por exemplo, podemos criar o arquivo `messages_en_US.properties`.

```
shoppingCart.title = Shopping cart
navigation.category.agile = Agile
navigation.category.front = Front end
navigation.category.games = Games
navigation.category.java = Java
navigation.category.mobile = Mobile
navigation.category.web = Web
navigation.category.others = Others
```

Podemos também criar o arquivo com a extensão `_pt.properties`, para suportar o português padrão. E agora a parte boa: não precisamos fazer mais nada! Basta que façamos a troca da língua padrão do navegador, que o arquivo correto vai ser escolhido. A pergunta que fica é: como o Spring descobre qual idioma o navegador prefere?

Fig. 13.1: Header com a opção de língua

A resposta está no cabeçalho da requisição HTTP. Perceba que tem uma chave chamada `Accept-Language` e é nela que vêm as línguas prediletas do seu usuário. No caso do navegador do autor deste livro, está como português. Tanto que a primeira opção é o **pt**, mas caso nosso sistema não suporte esse idioma, ele fala que prefere o **en-US**. Carregar o arquivo de mensagens em função da língua do navegador é a maneira padrão de o Spring MVC lidar com a internacionalização.

13.3 Passando parâmetros nas mensagens

Outra situação muito comum no momento de exibir as mensagens internacionalizadas é a necessidade de que a mensagem seja construída dinamicamente. Por exemplo, na página de listagem de livros, queremos dar boas-vindas ao usuário que acabou de logar no sistema.

```
Olá ${user.name}
```

Vamos deixar inclusive essa mensagem internacionalizada. Para isso, vamos acrescentar a entrada no arquivo de mensagens.

```
users.welcome = Olá {0}
```

Para representar os parâmetros, usamos os índices. Esta técnica já vem suportada por padrão no suporte à internacionalização presente no próprio Java. Agora, para usar na página, podemos usar a tag do Spring.

```
<spring:message code="users.welcome" arguments="${user.name}"/>
```

No atributo `arguments` passamos todos os valores que deverão substituir os índices na mensagem. Caso tenhamos mais de um, podemos separá-los por `,`. Para o mesmo processo ser feito usando a JSTL, teríamos algo parecido com o que segue:

```
<fmt:message key="users.welcome">
    <fmt:param value="${user.name}"/>
</fmt:message>
```

O primeiro modo é bem mais direto, por isso optamos por ele neste momento. Lembre-se de adicionar as entradas nos arquivos das duas línguas pois, caso você busque por uma chave inexistente, com a tag do Spring, uma exception vai ser lançada.

13.4 Deixe que o usuário defina a língua

Respeitar o idioma sugerido pelo navegador é uma ótima forma de tentarmos acertar a língua preferida pelo usuário. O problema dessa abordagem é se o idioma configurado não for realmente o preferido dele. Nesse momento, caso você esteja achando que todo mundo pode trocar isso facilmente, pense em todas as pessoas que você já conheceu que não tinham muito conhecimento de computação. Meu pai por exemplo, apesar de usar diariamente um navegador, não saberia como fazer essa troca.

Uma estratégia adotada por alguns sites é a de oferecer links para que o próprio usuário possa definir o idioma predileto.

Fig. 13.2: Bandeiras dos países

Vamos adotar a mesma abordagem para a Casa do Código. Inicialmente, vamos adicionar os links para que o usuário possa escolher. Faremos isso no

arquivo `header.jsp`:

```
<nav id="main-nav">
    <ul id="clearfix">
        ...
        <li>
            <a href="<c:url value="/produtos?locale=pt"/>">
                Português
            </a>
        </li>
        <li>
            <a href="<c:url value="/produtos?locale=en_US"/>">
                Inglês
            </a>
        </li>
    </ul>
</nav>
```

Já até passamos um parâmetro indicando os idiomas que queremos. A pergunta que fica é: quem vai receber esse parâmetro e trocar a língua preferida? De novo, não precisamos nos preocupar, o Spring MVC já vem com o suporte pronto.

```
public class AppWebConfiguration extends WebMvcConfigurerAdapter{

    ...

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new LocaleChangeInterceptor());
    }

    @Bean
    public LocaleResolver localeResolver(){
        return new CookieLocaleResolver();
    }

    ...
}
```

O método `localeResolver` retorna uma implementação da interface `LocaleResolver`. No nosso caso, optamos por usar a estratégia que guarda o idioma preferido em um cookie. Existem outras, como a `SessionLocaleResolver`, que permite que a preferência fique guardada diretamente na sessão do usuário. Além disso, precisamos ensinar ao Spring MVC que ele deve trocar o valor da língua em função do parâmetro passado.

Uma maneira padrão de fazer isso seria criando um `@Controller`. Só que, em vez de usar essa estratégia, vamos usar um `interceptor` já pronto, fornecido pelo Spring MVC. Interceptors funcionam como filtros, só que dentro do framework. A ideia é que eles possam ser executados antes ou depois da execução de algum método dos nossos controllers. Uma aplicação comum deles é quando o programador decide fazer o processo de autenticação na mão. Como ele tem que verificar se o usuário está logado para toda requisição, ele cria um interceptor para fazer essa checagem antes dos métodos dos controllers. No nosso caso, o `LocaleChangeInterceptor` verifica se foi usado o parâmetro `locale` na requisição e, em caso positivo, ele efetua a troca do idioma. Segue o seu código, apenas para curiosidade:

```
public class LocaleChangeInterceptor extends
HandlerInterceptorAdapter {

/**
 * Default name of the locale specification parameter: "locale".
 */
    public static final String DEFAULT_PARAM_NAME = "locale";

    private String paramName = DEFAULT_PARAM_NAME;


    public void setParamName(String paramName) {
        this.paramName = paramName;
    }

    public String getParamName() {
        return this.paramName;
    }
```

```
    @Override
    public boolean preHandle(HttpServletRequest request,
    HttpServletResponse response, Object handler)
            throws ServletException {

        String newLocale = request.getParameter(this.paramName);
        if (newLocale != null) {
            LocaleResolver localeResolver =
                RequestContextUtils.getLocaleResolver(request);
            if (localeResolver == null) {
                throw new IllegalStateException
                    ("No LocaleResolver found: not
                    in a DispatcherServlet request?");
            }
            localeResolver.setLocale(request, response,
             StringUtils.parseLocaleString(newLocale));
        }
        // Proceed in any case.
        return true;
    }

}
```

O método `preHandler`, como o próprio nome diz, é invocado antes da execução do nosso controller. Um ponto que deve ser lembrado é: a partir do momento em que o cookie do usuário foi setado, o Spring MVC sempre dará preferência a ele em detrimento do header `Accept-Language`.

13.5 Conclusão

Este capítulo também foi tranquilo. Internacionalização é um tema muito comum entre todas as aplicações, lembre-se apenas de não ser radical. Caso você não tenha planos de fazer outras versões, não se preocupe com este detalhe. Só o use se realmente for necessário, afinal de contas você está deixando de dar manutenção só em um lugar para dar em dois! Antes era apenas a página, agora é página e arquivo de mensagens.

Para o leitor mais esquecido, o carregamento do arquivo de mensagens já foi configurado bem no início do livro.

```
@Bean
public MessageSource messageSource() {
    ReloadableResourceBundleMessageSource bundle =
        new ReloadableResourceBundleMessageSource();
    bundle.setBasename("/WEB-INF/messages");
    bundle.setDefaultEncoding("UTF-8");
    bundle.setCacheSeconds(1);
    return bundle;
}
```

Capítulo 14

Testes automatizados

Nosso sistema já está com as principais funcionalidades implementadas. Cadastramos e listamos os livros, adicionamos produtos ao carrinho de compras e até simulamos integração com um sistema externo. Podemos ainda adicionar mais coisas, como só listar os livros que foram aprovados, colocar livros em destaque, associá-los às respectivas categorias etc. Você já é capaz de implementar tudo isso. Um detalhe muito importante, que não foi tratado até este momento, é a parte de testes da nossa aplicação. Não queremos ficar rodando tudo manualmente para saber se as coisas estão funcionando.

Existem algumas categorias de teste, geralmente divididas entre testes de unidade, de integração e de aceitação. A primeira categoria não vai ser abordada por este livro pois o Spring não tem nada de novo para oferecer nessa área. Você continuará usando o **JUnit** da mesma maneira como já deve ter lido em vários lugares. Em relação aos testes de aceitação, o Spring também não oferece nada de novo, você continuará trabalhando com ferramentas no

estilo do Selenium. Para uma base maior sobre a teoria relacionada a cada tipo de teste, aconselho o livro *TDD no Mundo Real*, do especialista e ótimo autor Maurício Aniche.

14.1 Testes de integração no DAO

A única categoria que sobrou foi a de teste de integração, mas não se engane: você vai passar uma boa parte do tempo trabalhando nele. Por exemplo, como saber se suas *queries* nos seus DAOs estão realmente funcionando? Geralmente, os códigos que necessitam de alguma infraestrutura, por exemplo do acesso ao banco de dados, exigem que você faça bastante configuração e simule cenários, dos quais o seu framework já cuida para você. Para esses tipos de testes, o Spring pode lhe ajudar muito.

Vamos começar adicionando um relatório na nossa aplicação. Precisamos saber o total vendido para cada tipo de livro.

```
@Repository
public class ProductDAO {

    @PersistenceContext
    private EntityManager manager;

    ...
    public BigDecimal sumPricesPerType(BookType bookType) {
        TypedQuery<BigDecimal> query = manager.createQuery(
                "select sum(price.value) from Product p join
                p.prices price where price.bookType =:bookType",
                BigDecimal.class);
        query.setParameter("bookType", bookType);
        return query.getSingleResult();
    }
}
```

Por mais que você confie nas suas habilidades como programador, sempre é interessante saber se a consulta está funcionando como você esperaria. Para verificar isso, podemos criar um teste que efetivamente cadastra alguns livros e verifica o retorno. Vamos criar a classe `ProductDAOTest`

na mesma estrutura de pacote já existente, só que vamos colocá-la dentro do `src/test/java`.

```
package br.com.casadocodigo.loja.daos;
public class ProductDAOTest {

    @Test
    public void shouldSumAllPricesOfEachBookPerType(){
        ProductDAO dao = new ProductDAO();

        //salva uma lista de livros impressos
        List<Product> printedBooks = ProductBuilder.
            newProduct(BookType.PRINTED, BigDecimal.TEN)
            .more(4).buildAll();
        //foreach do Java8, fique à vontade para usar um for
        //normal
        printedBooks.stream().forEach(productDAO::save);

        //salva uma lista de ebooks
        List<Product> ebooks = ProductBuilder.
            newProduct(BookType.EBOOK, BigDecimal.TEN)
            .more(4).buildAll();
        //foreach do Java8, fique à vontade para usar um for
        //normal
        ebooks.stream().forEach(productDAO::save);


        BigDecimal value =
            dao.sumPricesPerType(BookType.PRINTED);
        Assert.assertEquals(new BigDecimal(50).setScale(2),
                            value);
    }
}
```

Não se preocupe com a classe `ProductBuilder`, ela foi criada apenas para isolar a criação dos livros. Mais tarde damos uma olhada nela com mais carinho. O ponto importante é que salvamos alguns livros, depois invocamos o método que faz a soma das vendas e verificamos o resultado no final. Neste exato momento, o código não deve nem estar compilando, já que não

importamos o JUnit para nosso projeto. Basta que seja adicionada mais uma dependência no `pom.xml`.

```
...

<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
</dependency>
```

Com tudo configurado, já podemos rodar os testes. O problema é que na hora da execução recebemos um sonoro `NullPointerException`.

Fig. 14.1: ProductDAO não recebeu o EntityManager injetado

O problema está dentro método `save` e não tem escapatória, o nosso `EntityManager` está nulo. Para o nosso DAO funcionar, um `EntityManager` deve ser injetado pelo Spring, só que fomos nós que instanciamos o objeto. Como o Spring vai controlar as dependências de um objeto se é o programador que o está criando?

O mais interessante nesse tipo de cenário é usar o próprio Spring e criar os objetos através dele. Inclusive, vamos ter que controlar a transação para garantir que os objetos sejam inseridos antes de executarmos as *queries*.

```
@Test
public void shouldSumAllPricesOfEachBookPerType() {
    AnnotationConfigApplicationContext ctx =
        new AnnotationConfigApplicationContext(
            JPAConfiguration.class, ProductDAO.class);
    productDAO = ctx.getBean(ProductDAO.class);

    PlatformTransactionManager txManager =
        ctx.getBean(PlatformTransactionManager.class);
    TransactionStatus transaction = txManager
        .getTransaction(new DefaultTransactionAttribute());
    //até aqui foi configuração do spring

    List<Product> printedBooks = ProductBuilder
            .newProduct(BookType.PRINTED, BigDecimal.TEN).more(4)
            .buildAll();
    printedBooks.stream().forEach(productDAO::save);

    List<Product> ebooks = ProductBuilder
            .newProduct(BookType.EBOOK, BigDecimal.TEN)
            .more(4).buildAll();
    ebooks.stream().forEach(productDAO::save);
    BigDecimal value =
        productDAO.sumPricesPerType(BookType.PRINTED);
    Assert.assertEquals(new BigDecimal(50).setScale(2), value);


    txManager.commit(transaction);
    ctx.close();
}
```

Não fique muito assustado com esse código, ele só demonstra um pouco do que acontece por trás dos panos quando você está utilizando o Spring MVC. Tivemos que criar o contexto do Spring na mão e também usamos os objetos dele para recuperar os objetos e controlar a transação. O ponto aqui

é: já temos muitas preocupações com nossos testes para ainda ter que ficar aprendendo a lidar com os detalhes internos do framework. Para não ficar preso nesse tipo de situação, o Spring criou um módulo focado apenas em ajudar nesses tipos de testes.

Configuração do Spring Test

Como já é de praxe, precisamos adicionar as dependências no `pom.xml`.

```
...
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-test</artifactId>
        <version>4.1.0.RELEASE</version>
        <scope>test</scope>
    </dependency>
```

Agora vamos partir para realizar o mesmo teste, só que trocando aquele código pelas annotations providas pelo Spring Test.

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes =
    {ProductDAO.class,JPAConfiguration.class})
public class ProductDAOTest {

    @Autowired
    private ProductDAO productDAO;

    @Transactional
    @Test
    public void shouldSumAllPricesOfEachBookPerType(){
        List<Product> printedBooks = ProductBuilder
            .newProduct(BookType.PRINTED, BigDecimal.TEN)
            .more(4).buildAll();
        printedBooks.stream().forEach(productDAO::save);

        List<Product> ebooks = ProductBuilder
            .newProduct(BookType.EBOOK, BigDecimal.TEN)
            .more(4).buildAll();
```

```
        ebooks.stream().forEach(productDAO::save);

        BigDecimal value =
            productDAO.sumPricesPerType(BookType.PRINTED);
        Assert.assertEquals(new BigDecimal(50).setScale(2),
            value);
    }
}
```

Respire, vamos desvendar essas novas annotations. O código do teste ficou muito parecido com o inicial. A principal mudança é que agora recebemos o `ProductDAO` injetado e também podemos usar a já conhecida `@Transactional` para dizer que nosso método precisa rodar dentro de uma transação.

A parte mais nova são as annotations que foram usadas em cima da nossa classe. A `@RunWith` é específica do JUnit e é fornecida para que frameworks possam ser notificados das fases de execução do teste. Por exemplo, a extensão do Spring Test é notificada da criação da instância do teste em si e, a partir disso, consegue fazer todas as injeções necessárias. E aí entra a outra pergunta: quais classes devem ser carregadas? É justamente para isso que serve a annotation `@ContextConfiguration`. Repare que passamos diretamente a classe `ProductDAO` e além disso passamos a `JPAConfiguration`, que contém configuração de todos objetos necessários para utilizarmos a JPA.

Um último ponto, que na verdade é o mais importante, é que nosso teste ainda está falhando. O problema é que estamos inserindo os produtos na tabela que vem sendo usada para desenvolvimento e lá, por consequência, já existem mais livros do que o planejado.

14.2 Utilize profiles e controle seu ambiente

Precisamos criar um ambiente específico para os testes, de forma que tenhamos controle absoluto sobre o que existe no banco de dados, para uma operação não atrapalhar a outra. A primeira ação que devemos realizar é a de criar um *Datasource* que aponte para outro banco de dados. Podemos criar uma classe de configuração só para isso, lembre-se que estamos trabalhando no *source folder* de testes.

```
package br.com.casadocodigo.loja.conf;

//imports

public class DataSourceConfigurationTest {

   @Bean
   public DataSource dataSource(){
       DriverManagerDataSource dataSource =
           new DriverManagerDataSource();
       dataSource.setDriverClassName("com.mysql.jdbc.Driver");
       dataSource.setUrl(
           "jdbc:mysql://localhost:3306/casadocodigo_test");
       dataSource.setUsername("root");
       dataSource.setPassword("");
       return dataSource;
   }
}
```

Agora basta que adicionemos esta classe na lista de classe carregadas:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes =
    {DataSourceConfigurationTest.class ,
    ProductDAO.class,JPAConfiguration.class})

public class ProductDAOTest {
    ...
}
```

Mesmo com essa alteração, quando rodamos o nosso teste, vemos que ele está com o mesmo problema do anterior. O resultado final ainda está sendo influenciado pelas tabelas usadas em ambiente de desenvolvimento. Só que nossa nova classe foi adicionada ao inicio na lista e, dessa forma, o `Datasource` configurado dentro da classe `JPAConfiguration` fica sendo o último a ser registrado.

Claro, podemos trocar a ordem, mas não precisa nem pensar muito para saber que isso vai dar uma confusão grande. O que queremos é configurar que o `Datasource` de testes deva ser carregado em um contexto de testes

enquanto o outro deve ser carregado em ambiente de desenvolvimento. Para fazer isso, podemos usar o recurso de **Profiles** do Spring.

```
public class DataSourceConfigurationTest {

   @Bean
   @Profile("test")
   public DataSource dataSource(){
       DriverManagerDataSource dataSource =
           new DriverManagerDataSource();
       dataSource.setDriverClassName("com.mysql.jdbc.Driver");
       dataSource.setUrl(
           "jdbc:mysql://localhost:3306/casadocodigo_test");
       dataSource.setUsername( "root" );
       dataSource.setPassword( "" );
       return dataSource;
   }
}
```

Agora, na classe `JPAConfiguration`, podemos dizer que o outro `Datasource` deve ser utilizado no ambiente de desenvolvimento.

```
@Bean
@Profile("dev")
public DataSource dataSource(){
    DriverManagerDataSource dataSource =
        new DriverManagerDataSource();
    dataSource.setDriverClassName("com.mysql.jdbc.Driver");
    dataSource.setUrl(
        "jdbc:mysql://localhost:3306/casadocodigo");
    dataSource.setUsername( "root" );
    dataSource.setPassword( "" );
    return dataSource;
}
```

Por fim, no nosso teste informamos que o profile ativo é o de `test`.

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes =
    {DataSourceConfigurationTest.class
```

```
    ,ProductDAO.class,JPAConfiguration.class})
@ActiveProfiles("test")
public class ProductDAOTest {
    ...
}
```

Como o próprio nome informa, a annotation `@ActiveProfiles` pode ser usada para indicar qual profile está ativo. Ela casa muito bem com a parte de testes, por isso vem dentro do Spring Test. Dessa forma, conseguimos deixar nosso teste passando! Além disso, já ganhamos um brinde: o Spring Test automaticamente limpa o banco de dados a cada teste. Isso já nos libera de ter que escrever este tipo de código e, além disso, não vamos ter um teste influenciando o outro.

Ative o profile para a aplicação web

Neste exato momento, nossa aplicação web parou de funcionar. Como não definimos nenhum profile nela, o `Datasource` não consegue ser encontrado. Para consertamos isso, podemos configurar o profile através de um parâmetro que deve ser lido no inicio do servidor. Podemos fazê-lo na classe `ServletSpringMVC`.

```
@Override
public void onStartup(ServletContext servletContext)
        throws ServletException {
    super.onStartup(servletContext);
    servletContext.addListener(RequestContextListener.class);
    servletContext.setInitParameter("spring.profiles.active",
    "dev");
}
```

Outra possibilidade seria fazer a configuração diretamente no `web.xml`.

```
<context-param>
    <param-name>spring.profiles.active</param-name>
    <param-value>live</param-value>
</context-param>
```

14.3 Testes do controller

Testar os DAOs, além dos testes de unidades em si, já fornece uma boa confiança à nossa base de código. Afinal de contas, a maioria das aplicações é bem focada no uso do banco de dados. Outro componente que talvez valha a pena ser testado é o nosso controller.

Geralmente, eles não vão possuir tanta lógica, basicamente vão controlar fluxos de chamadas para outros objetos. Mesmo sendo até certo ponto triviais de serem entendidos, alguns controllers não podem funcionar de maneira equivocada. Por exemplo, a URL de produtos deve sempre listar os livros

que estão no banco de dados, caso contrário a Casa do Código pode estar perdendo vendas.

Os controllers são acessíveis através das URLs e para que isso funcione precisamos que o servidor esteja de pé. Este teste até pode ser feito e geralmente é usada uma biblioteca como o Selenium para ajudar em todo o processo. O ponto negativo dessa abordagem é justamente precisar esperar todo o tempo de start do servidor. O Spring Test pode nos ajudar aí. Ele consegue montar o ambiente necessário para que consigamos acessar os controllers através das URLs, sem a necessidade de ter um servidor rodando.

```
package br.com.casadocodigo.loja.controllers;

//imports

@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration(classes = { AppWebConfiguration.class,
        JPAConfiguration.class, SecurityConfiguration.class,
        DataSourceConfigurationTest.class })
@ActiveProfiles("test")
public class ProductsControllerTest {
    @Autowired
    private ProductDAO productDAO;

    @Autowired
    private WebApplicationContext wac;

    private MockMvc mockMvc;

    @Before
    public void setup() {
        this.mockMvc =
            MockMvcBuilders.webAppContextSetup(this.wac)
            .build();
    }

    @Test
    @Transactional
```

```
    public void shouldListAllBooksInTheHome() throws Exception {

        this.mockMvc.perform(get("/produtos"))
        .andExpect(MockMvcResultMatchers
        .forwardedUrl("/WEB-INF/views/products/list.jsp"));
    }
```

A parte importante desse código é a simulação do request. Perceba que conseguimos fazer um `get` e dizer que esperamos que tenha sido gerado um `forward` para a JSP de listagem dos produtos. Por exemplo, podemos ir além e checar se efetivamente foi adicionada uma variável com nome `products`, para ficar disponível na página.

```
this.mockMvc.perform(get("/produtos"))
.andExpect(modelAndViewMatcher)
.andExpect(MockMvcResultMatchers
.model()
.attributeExists("products"))
.andExpect(MockMvcResultMatchers
            .forwardedUrl("/WEB-INF/views/products/list.jsp"));
```

Esse código é suficiente para fazer nosso teste passar. Vamos apenas analisar esse trecho um pouco mais, para entender melhor o que o método `andExpect` espera como argumento.

Matchers

```
ResultActions andExpect(ResultMatcher matcher) throws Exception;
```

O método `andExpect` espera como parâmetro um objeto do tipo `ResultMatcher`, cujas implementações simplesmente fazem uma verificação para a gente. São objetos desse tipo que são retornados pelos métodos `attributeExists(...)` ou `forwardedUrl(...)`.

```
public ResultMatcher attributeExists(final String... names) {
    return new ResultMatcher() {
        @Override
        public void match(MvcResult result) throws Exception {
```

```
            ModelAndView mav = getModelAndView(result);
            for (String name : names) {
                assertTrue("Model attribute '" + name +
                "' does not exist", mav.getModel().get(name)
                != null);
            }
        }
    };
}
```

Caso você precise de alguma checagem que não existe pronta, não fique acanhado, vá lá e crie o seu `ResultMatcher`.

Objetos e annotations de configuração

Voltando um pouco para o setup do teste, vamos dar uma olhada nos objetos e annotations necessários para fazer o nosso código funcionar.

```
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration(classes = { AppWebConfiguration.class,
        JPAConfiguration.class, SecurityConfiguration.class,
        DataSourceConfigurationTest.class })
@ActiveProfiles("test")
public class ProductsControllerTest {
    @Autowired
    private ProductDAO productDAO;

    @Autowired
    private WebApplicationContext wac;

    private MockMvc mockMvc;

    @Before
    public void setup() {
        this.mockMvc =
            MockMvcBuilders.webAppContextSetup(this.wac)
            .build();
    }
```

Em relação às annotations, a única novidade é a `@WebAppConfiguration`. Ela é responsável por fazer o *Runner* do Spring Test carregar os objetos necessários para uma aplicação web, por exemplo, o `HttpServletRequest` e o `HttpServletResponse`. Além disso, precisamos criar o objeto responsável por simular os requests para nós, no caso o `MockMvc`. O objeto do tipo `WebApplicationContext` é a representação do contexto do Spring para projetos web. Precisamos dele para montar o `MockMvc`, já que será necessário simular requests, response etc.

Testes para garantir o controle de acesso

Um último teste que podemos fazer é o que garante que só usuários com perfil de **ADMIN** possam acessar o endereço de cadastro de produtos.

```
@Test
public void onlyAdminShoudAccessProductsForm()
        throws Exception {
    RequestPostProcessor processor =
            SecurityMockMvcRequestPostProcessors
            .user("comprador@gmail.com").password("123456")
            .roles("COMPRADOR");
    this.mockMvc.perform(get("/produtos/form")
    .with(processor))
    .andExpect(MockMvcResultMatchers.status().is(403));
}
```

Funciona basicamente no mesmo estilo do anterior, a única diferença é que usamos uma classe específica do módulo de testes do Spring Security. Pedimos para que seja criado um usuário com as características de que a gente precisa e tentamos acessar a URL protegida. Deve ser retornado o status 403, informando que aquele usuário não tem autorização para acessar determinada URL. O objeto do tipo `RequestPostProcessor` pode ser usado quando precisamos adicionar alguma coisa ao request antes de ele ser processado. É exatamente isso que é feito e um objeto que representa nosso usuário é adicionado, descartando a necessidade de termos que ir ao banco de dados fazer a busca.

Como quase sempre, vamos para as configurações finais. O filtro do Spring Security deve ser adicionado ao `MockMvc` para que ele possa interceptar o request e aplicar as regras necessárias.

```
@ContextConfiguration(classes = { AppWebConfiguration.class,
        JPAConfiguration.class, SecurityConfiguration.class,
        DataSourceConfigurationTest.class })
@ActiveProfiles("test")
public class ProductsControllerTest {

    @Autowired
    private Filter springSecurityFilterChain;

    @Before
    public void setup() {
        this.mockMvc =
            MockMvcBuilders.webAppContextSetup(this.wac)
            .addFilters(springSecurityFilterChain).build();
    }
```

Recebemos injetado o filtro, já que ele é carregado através da nossa classe `SecurityConfiguration` e o adicionamos à simulação da nossa aplicação web. Para que essa parte funcione, precisamos adicionar mais uma dependência ao nosso `pom.xml`.

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-test</artifactId>
    <version>4.0.0.M2</version>
    <scope>test</scope>
</dependency>
```

14.4 Conclusão

Testar é uma parte essencial da nossa aplicação e os testes de integração têm um papel bem importante nisso. Com a ajuda fornecida pelo Spring Test, tudo fica muito mais simples, você tem que focar apenas na escrita do teste em si.

Outro ponto a ser lembrado deste capítulo é a utilização das annotations de profile. Elas são de grande ajuda tanto no cenário de testes quanto em relação ao uso de objetos com ligação com o mundo exterior. Por exemplo, você não vai ficar querendo fazer upload de arquivos para a Amazon enquanto está em desenvolvimento e, para contornar isso, pode usar um profile que carregue um objeto que fique salvando no disco local. Existem várias aplicações, fique sempre atento.

Capítulo 15

Outras facilidades

Já passamos por bastante coisa no decorrer do livro, mas para fechar com chave de ouro vamos ver mais facilidades providas pelo Spring que, com certeza, serão úteis no seu dia a dia de usuário do framework.

15.1 Resolvendo o problema gerado pelo Lazy Load

Não sei se você está lembrado, mas na hora da escrita dos métodos da nossa classe `ProductDAO` usamos a expressão `join fetch` para ficar carregando a lista de preços relativas a um livro.

```
public List<Product> list() {
    return manager.createQuery(
            "select p from Product p join fetch p.prices",
```

```
            Product.class).getResultList();
}
```

Só fazemos isso porque na nossa JSP tentamos carregar essa mesma lista. Olhe o código a seguir para refrescar a memória.

```
<c:forEach items="${product.prices}" var="price">
    [${price.value} - ${price.bookType}]
</c:forEach>
```

Caso você não utilize o `fetch join` vai receber a seguinte exception:

```
org.hibernate.LazyInitializationException: failed to lazily
initialize a collection of role:
br.com.casadocodigo.loja.models.Product.prices
```

Ela informa que o Hibernate não conseguiu carregar a coleção de preços. Essa é a parte boa e ruim de usar um framework como o Hibernate. Quando você carrega seus objetos, ele substitui algumas propriedades para permitir que você programe de maneira orientada a objetos. Por exemplo basta invocar o método `getPrices()` que ele vai tentar disparar uma query no banco. O ponto ruim é que você precisa entender um pouco mais dele, para conseguir tirar proveito dessa funcionalidade. Basicamente, o `EntityManager` tem que ficar aberto durante todo o ciclo da requisição, inclusive no momento em que você está montando a JSP.

Para facilitar nossa vida, o módulo do Spring de integração com a JPA e Hibernate já fornece um filtro, da API de Servlets, que faz esse trabalho para a gente.

```
public class ServletSpringMVC extends
    AbstractAnnotationConfigDispatcherServletInitializer {
        ....

        @Override
        protected Filter[] getServletFilters() {
            return new Filter[]{
                new OpenEntityManagerInViewFilter()};
        }
}
```

Basta que façamos o registro do filtro que agora você pode invocar os métodos que você quiser, o `EntityManager` vai estar sempre aberto. Não estranhe que estamos criando o filtro na mão, desde a versão 3 da especificação de Servlets, podemos registrar quase tudo de maneira programática.

Caso você tire o `fetch join` da consulta e acesse a tela de listagem de produtos, vai perceber que tudo acontecerá sem problema algum. Caso você seja um pouco mais curioso e analise o que saiu no console do servidor, verá algo como o exemplo a seguir.

```
Hibernate: select prices0_.Product_id as Product_1_0_0_,
    prices0_.bookType as bookType2_1_0_, prices0_.value as
    value3_1_0_ from Product_prices
    prices0_ where prices0_.Product_id=?

Hibernate: select prices0_.Product_id as Product_1_0_0_,
    prices0_.bookType as bookType2_1_0_, prices0_.value as
    value3_1_0_ from Product_prices prices0_ where
    prices0_.Product_id=?

Hibernate: select prices0_.Product_id as Product_1_0_0_,
    prices0_.bookType as bookType2_1_0_, prices0_.value as
    value3_1_0_ from Product_prices prices0_ where
    prices0_.Product_id=?
...
```

Para cada produto carregado, o Hibernate fez uma query no banco para buscar os preços necessários. Esse é o problema das *N + 1* queries causado pelo uso do Lazy Load. Na opinião deste autor, você deveria tentar planejar o máximo de queries que pudesse e deixar para usar o Lazy Load só em último caso. Inicialmente vai parecer que ele está ajudando muito, mas com o passar do tempo você vai começar a perceber que muitas consultas estão sendo realizadas sem que você tenha controle.

15.2 Liberando acesso a recursos estáticos da aplicação

Continuando com as dicas de final de livro. Para montar as telas da nossa aplicação, em alguns momentos, copiamos o `HTML` gerado pelo site da Casa do Código. Adicionamos o Spring Security na nossa aplicação e tudo continuou funcionando da maneira como deveria. Uma questão para pensarmos é: e se tivéssemos adicionado arquivos de Javascript e de CSS, o que aconteceria?

Vamos fazer um teste simples. Vou criar um novo arquivo `.js` na pasta `src/main/webapp/resources/js`. Para simular uma situação real, podemos criar o arquivo com o nome de `jquery.js`. Agora iniciamos o servidor e tentamos acessá-lo através da URL localhost:8080/casadocodigo/resources/js/jquery.js. Você vai perceber que prontamente o Spring Security o direcionou para a tela de login. Para o framework, é apenas mais uma URL e, como não fizemos a configuração liberando o acesso a ela, somos bloqueados.

Para resolver isso, vamos sobrescrever um outro método de configuração.

```
@EnableWebMvcSecurity
public class SecurityConfiguration extends
    WebSecurityConfigurerAdapter{

    ...

    @Override
    public void configure(WebSecurity web) throws Exception {
        web.ignoring().antMatchers("/resources/**");
    }
}
```

Ao contrário do primeiro `configure`, que recebe como argumento um objeto do tipo `HttpSecurity`, esse recebe um `WebSecurity`. O primeiro é utilizado para fazer a configuração geral. Como vimos, ele faz muito mais coisas do que apenas permitir o acesso a algumas URLs. Este que estamos vendo agora é específico para a aplicação web. Por exemplo, estamos pedindo para o Spring Security ignorar qualquer acesso a URL que comece com `resources`.

Agora, quando você tentar acessa a mesma URL, o servidor vai retornar o status 404. O problema é que a Servlet do Spring MVC está achando que você está acessando um endereço mapeado para um controller, o que claramente não é o caso. Para resolver isso, vamos sobrescrever mais um método, agora na nossa classe `AppWebConfiguration`.

```
@Override
public void configureDefaultServletHandling(
        DefaultServletHandlerConfigurer configurer) {
    configurer.enable();
}
```

Quando você invoca o método `enable()` na classe `DefaultServletHandlerConfigurer`, você está informando para o Spring MVC que, caso ele não consiga resolver o endereço, ele deve delegar a chamada para o Servlet Container em uso. Pronto, agora você já consegue servir os seus conteúdos estáticos. Lembre-se de deixá-los dentro de uma pasta específica, para ficar fácil de você configurar.

15.3 Enviando e-mail

Uma última tarefa que você vai acabar precisando realizar é a de enviar e-mails. Quase todo sistema tem uma funcionalidade da qual enviar um alerta via e-mail é parte integrante. Como não poderia ser diferente, o Spring já vem com a abstração necessária para que possamos enviar um e-mail de forma simples.

O sistema da Casa do Código, por exemplo, tem a situação ideal. Quando uma compra é realizada com sucesso, precisamos mandar um e-mail para o comprador.

```
@RequestMapping("/payment")
public class PaymentController {
    ...
    @Autowired
    private MailSender mailer;

    @RequestMapping(value = "checkout",
```

```
                    method = RequestMethod.POST)
    public Callable<ModelAndView> checkout(
        @AuthenticationPrincipal User user) {

        return () -> {
            BigDecimal total = shoppingCart.getTotal();
            String uriToPay =
                "http://book-payment.herokuapp.com/payment";
            try {
                restTemplate.postForObject(uriToPay,
                        new PaymentData(total), String.class);
                //enviando email
                sendNewPurchaseMail(user);
                return new ModelAndView(
                    "redirect:/payment/success");
            } catch (HttpClientErrorException exception) {
                return new ModelAndView(
                    "redirect:/payment/error");
            }

        };
    }

    private void sendNewPurchaseMail(User user) {
        SimpleMailMessage email = new SimpleMailMessage();
        email.setFrom("compras@casadocodigo.com.br");
        email.setTo(user.getLogin());
        email.setSubject("Nova compra");
        email.setText("corpo do email");
        mailer.send(email);
    }
}
```

Perceba que apenas recebemos um objeto do tipo `MailSender`, que é uma interface do Spring, e mandamos o e-mail. Mais simples, impossível. De brinde, ainda usamos a annotation `@AuthenticationPrincipal`, que permite que recebamos o usuário logado como parâmetro do nosso método. Pensando em design de código, poderíamos ter criado outra classe unicamente responsável por enviar e-mails no sistema e tê-la usado no nosso con-

troller. Essa é uma tarefa que fica para você!

Para o Spring conseguir fazer a injeção, precisamos ensinar-lhe qual implementação será usada. Para fazermos isso, mais uma vez, criaremos um método na nossa classe `AppWebConfiguration`.

```
@Bean
public MailSender mailSender() {

    JavaMailSenderImpl javaMailSenderImpl =
        new JavaMailSenderImpl();
    javaMailSenderImpl.setHost("smtp.gmail.com");
    javaMailSenderImpl.setPassword("password");
    javaMailSenderImpl.setPort(587);
    javaMailSenderImpl.setUsername("seulogin");
    Properties mailProperties = new Properties();
    mailProperties.put("mail.smtp.auth", true);
    mailProperties.put("mail.smtp.starttls.enable", true);
    javaMailSenderImpl.setJavaMailProperties(mailProperties);

    return javaMailSenderImpl;
}
```

Os métodos são bem legíveis. A única coisa a mais, é a configuração do `Properties`. Alguns provedores de e-mail, como o gmail, vão pedir que você faça uma conexão em modo seguro, por isso a habilitação do **Transport Layer Security(TLS)**.

Para finalizar, como não poderia deixar de ser, vamos adicionar as dependências necessárias para que o Spring possa utilizar a especificação de e-mail, provida pelo próprio Java.

```
<!-- email -->

<dependency>
    <groupId>javax.mail</groupId>
    <artifactId>mail</artifactId>
    <version>1.4.7</version>
</dependency>
```

15.4 Conclusão

Essas últimas facilidades foram apenas para mostrar que, mesmo passando por muita coisa ao decorrer do livro, ainda tivemos mais detalhes para mostrar. Tome bastante cuidado com a decisão de usar o Hibernate no projeto porque, por mais que ele abstraia muito da complexidade da persistência, ele vai exigir que você entenda mais a fundo o funcionamento dele, justamente para não cair nas armadilhas. Vimos também que é importante configurar nossos frameworks para que eles não bloqueiem nossos recursos estáticos, como Javascript e CSS.

Não pare agora e já leia o próximo capítulo. Vamos fazer as configurações necessárias para realizar o deploy da nossa aplicação.

Capítulo 16

Deploy da aplicação

Trabalhamos tanto durante o livro, que seria injusto você acabá-lo e não poder mostrar para ninguém o que desenvolveu. Para resolvermos isso, vamos instalar nossa aplicação no **Heroku**, uma plataforma que suporta o deploy de aplicações das mais variadas linguagens.

16.1 Configurando o Maven para o Heroku

Ao contrário da maioria dos hosts, o Heroku não pede que você suba o arquivo `.war` da sua aplicação para o Tomcat ou Jetty dele. Em vez disso, é necessário usarmos uma outra versão do Tomcat, chamada de `embedded`. A diferença principal é que ele não suporta múltiplos contextos. Simplesmente o rodamos, pode até ser a partir de um método `main` e passamos a localização do nosso `war`.

```
java -jar target/dependency/webapp-runner.jar target/*.war
```

O `jar webapp-runner` é que o vamos baixar para conseguir executar a versão minimalista do servidor. Vamos ensinar ao Maven que, quando ele for gerar o nosso arquivo de instalação, ele deve baixar o Tomcat *embedded* e copiar para o `jar webapp-runner.jar`. Para isso, vamos alterar o arquivo `pom.xml`.

```
<dependencies>
    ...
</dependencies>

<build>
<plugins>

    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-dependency-plugin</artifactId>
        <version>2.3</version>
        <executions>
            <execution>
                <phase>package</phase>
                <goals>
                    <goal>copy</goal>
                </goals>
                <configuration>
                    <artifactItems>
                        <artifactItem>
                          <groupId>com.github.jsimone</groupId>
                          <artifactId>webapp-runner</artifactId>
                          <version>7.0.57.2</version>
                          <destFileName>
                              webapp-runner.jar
                          </destFileName>
                        </artifactItem>
                    </artifactItems>
                </configuration>
            </execution>
        </executions>
```

```
    </plugin>
</plugins>
</build>
```

Adicionamos um plugin do Maven que realiza a cópia necessária para nós. Esse procedimento está também descrito na própria página do Heroku. Basta seguir o link: https://devcenter.heroku.com/articles/java-webapp-runner.

Agora que o plugin está configurado, podemos pedir para o Maven gerar nosso `war`.

```
caminhoInstalacaoMaven/bin/mvn package
```

16.2 Qual banco será usado no Heroku

Nesse exato momento, temos dos profiles para carregar o `DataSource` com os dados de conexão do banco de dados. Um para o ambiente de testes e outro, que é o default, para o ambiente de desenvolvimento. E para o ambiente de produção, qual banco vamos usar? Esse também é um caso que a parte de profile pode nos ajudar.

O Spring permite que você crie uma variável de ambiente informando qual profile deve ser carregado pela aplicação. Outra maneira, mais simples, é passar um argumento na execução do programa, indicando que você quer deixar uma variável disponível, como se fosse uma de ambiente. Seguem as duas opções:

```
SPRING_PROFILES_ACTIVE=prod
    java -jar target/dependency/webapp-runner.jar target/*.war

    java -jar -Dspring.profiles.active=prod
        target/dependency/webapp-runner.jar  target/*.war
```

É importante que você execute esses comandos em apenas uma linha.

Definindo a variável, na hora de carregar o container, o Spring vai procurar por *beans* que estejam anotados com `@Profile("prod")`. Para que tudo funcione, podemos criar outra classe de configuração que contenha apenas a versão do `DataSource` apontando para o outro banco.

```
@Profile("prod")
public class JPAProductionConfiguration {

        @Autowired
        private Environment environment;

        @Bean
        public DataSource dataSource() throws URISyntaxException{
           DriverManagerDataSource dataSource =
               new DriverManagerDataSource();
           dataSource.setDriverClassName("org.postgresql.Driver");

           URI dbUrl =
               new URI(environment.getProperty("DATABASE_URL"));
           dataSource.setUrl("jdbc:postgresql://"
               + dbUrl.getHost() + ":"
               + dbUrl.getPort() + dbUrl.getPath());
           dataSource.setUsername(dbUrl.getUserInfo()
                                         .split(":")[0]);
           dataSource.setPassword(dbUrl.getUserInfo()
                                         .split(":")[1]);

           return dataSource;
        }
}
```

Eis um ponto que pode ter ficado na sua cabeça em relação ao banco usado para o ambiente de produção. O leitor mais atento deve ter notado que foi usado o PostgreSQL em vez do MYSQL. Fomos por este caminho apenas porque o Heroku usa o Postgre como default. De todo jeito, vale ressaltar que o melhor seria usar o mesmo banco em desenvolvimento e em produção. As informações de conexão foram extraídas da variável de ambiente fornecida pelo próprio Heroku. Segue um exemplo da formatação:

```
export
DATABASE_URL=postgres://user:password@localhost:5432/nomeDaApp
```

Podemos acessar qualquer variável de ambiente através da classe `Environment`, já disponibilizada pelo próprio Spring. Através dela também

podemos saber em qual ambiente estamos rodando, por exemplo.

Lembre-se que também precisamos adicionar essa nova configuração à classe `ServletSpringMVC`. A ideia é que ela será sempre carregada, só que vai ser usada apenas quando o profile de produção estiver ativo.

```
public class ServletSpringMVC extends
        AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class[] { SecurityConfiguration.class,
                AppWebConfiguration.class,
                JPAConfiguration.class,
                JPAProductionConfiguration.class };
    }

    ....
}
```

16.3 Nova aplicação no Heroku

Nossa aplicação já está devidamente configurada para ser deployada. Para finalizar, precisamos apenas criar uma nova aplicação no próprio Heroku.

O Heroku faz todo processo de deploy ser baseado no **git**, de modo que é imprescindível que você o tenha instalado em sua máquina. Caso não o tenha, siga os passos aqui: https://help.github.com/articles/set-up-git/. Com o git configurado, acesse a pasta da sua aplicação e execute os seguintes comandos:

```
git init

git add -A

git commit -m "versionando o código para deploy"
```

O próximo passo é instalar o programa de linha de comando fornecido pelo Heroku. Para realizar o download siga os passos sugeridos neste link: https://toolbelt.heroku.com/.

Com o **Heroku Toolbelt** instalado, podemos criar uma nova aplicação. Rode o comando de dentro da pasta da sua aplicação, dessa forma o utilitário já vai configurar o seu repositório, fazendo com que ele aponte para o Heorku.

```
heroku apps:create cdcspringmvc
```

Essa linha é suficiente para ser criada uma nova aplicação. Tome apenas cuidado com o nome, porque ele pode já ter sido usado. Para finalizar a configuração, precisamos ensinar ao Heroku qual linha ele deve rodar para iniciar a nossa aplicação. Para isso, na raiz do projeto, temos que criar um arquivo chamado `Procfile`, com o seguinte conteúdo, e tudo na mesma linha:

```
web:  java $JAVA_OPTS -jar -Dspring.profiles.active=prod
    target/dependency/webapp-runner.jar --port $PORT target/*.war
```

O `JAVA_OPTS` é apenas uma variável de ambiente definida pelo Heroku, cuja utilização é sugerida para subir a aplicação.

Agora é a hora de ser feliz e subir o seu código!

```
git push heroku master
```

Pronto, você acabou de realizar o primeiro deploy no Heroku. Esta é uma vantagem grande dos Clouds, tudo já está configurado e automatizado do lado do servidor, basta que você se adeque um pouco e terá um deploy tranquilo.

16.4 Conclusão

Neste capítulo, estudamos um pouco do processo de deploy da aplicação. Ainda conseguimos ver um pouco mais de como a parte de profile pode nos ajudar.

Duas lições de casa para você, apenas para deixar a aplicação ainda mais legal: é necessário que você crie um endereço para cadastro de novos usuários, para poder conseguir realizar o login da aplicação. Além disso, pode ser legal adicionar algum CSS para deixar o visual mais interessante.

Capítulo 17

Hora de praticar

Muito obrigado por ter ficado comigo durante toda a jornada do livro! É muito importante que você tente praticar tudo que foi estudado. Tem chance de escolher a tecnologia do novo projeto? Não fique acanhado, use o Spring MVC e foque nas suas regras de negócio entregando muito mais valor para o seu cliente. Sem contar que, mesmo este ditado sendo velho, a prática leva à perfeição. Em cada novo projeto, você vai encontrar novos desafios que farão com que você entenda cada vez mais sobre o framework.

17.1 Mantenha contato

Eu estou completamente disponível para ajudá-lo onde for possível. Você pode me achar das seguintes formas:

- Por e-mail, basta enviar uma mensagem para livrospringmvc@gmail.com
- Caso use o Twitter, é só acessar https://twitter.com/alberto_souza
- Ainda há a possibilidade de acompanhar meus projetos pelo Github, basta acessar https://github.com/asouza.
- Caso queira postar uma dúvida e ainda ajudar outras pessoas, use o fórum criado especialmente para o livro, http://groups.google.com.br/d/forum/livro-springmvc
- Vá além do conteúdo do livro e também acompanhe os posts feitos no blog https://domineospring.wordpress.com.

Por último, todo e qualquer feedback sobre o livro é bem-vindo. Use o fórum ou mande diretamente para meu e-mail. Escrever o livro foi um grande desafio, espero que ele tenha sido de alguma valia para você.